Grant Number NAG8-093

An Intelligent Allocation Algorithm For Parallel Processing

by

Chester C. Carroll
Cudworth Professor of Computer Architecture

Abdollah Homaifar
Temporary Visiting Assistant Professor
of Electrical Engineering

and

Kishan G. Ananthram
Graduate Research Assistant

Prepared for

The National Aeronautics and Space Administration

Bureau of Engineering Research
The University of Alabama
January 1988

BER Report No. 416-17

ABSTRACT

This research considers the problem of allocating nodes of a program graph, to processors, in a parallel processing architecture. The algorithm is based on critical path analysis, some allocation heuristics, and the execution granularity of nodes in a program graph. These factors, and the structure of interprocessor communication network, influence the allocation. To achieve realistic estimations of the execution durations of allocations, the algorithm considers the fact that nodes in a program graph have to communicate through varying numbers of tokens. Coarse and fine granularities have been implemented, with interprocessor token-communication duration, varying from zero up to values comparable with the execution durations of individual nodes. The effect on allocation, of communication network structures, is demonstrated by performing allocations for crossbar (non-blocking) and star (blocking) networks. The algorithm assumes the availability of as many processors as it needs for the optimal allocation of any program graph. Hence, the focus of allocation has been on varying token-communication durations rather than varying the number of processors. The algorithm always utilizes as many processors as necessary for the optimal allocation of any program graph, depending upon granularity and characteristics of the interprocessor communication network.

ACKNOWLEDGEMENT

This research was supported by NASA, George C. Marshall Space Flight Center, Huntsville, Alabama, under Grant Number NAG8-093 and conducted in the Computer Architecture Research Laboratory in the College of Engineering at The University of Alabama.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

PROBLEM STATEMENT

1.1 INTRODUCTION

With upper limits being reached in the speed of semiconductor technology, new avenues have to be exploited for achieving real time response to many practical problems. Parallel processing promises considerable insight into solutions for such applications. Extensive research is being conducted in this direction. Parallel processing is possible because of the existence of implicit parallel executions of the instructions and tasks of application algorithms.

The conventional Von-Neumann architecture employs program control to execute instructions sequentially. Parallel processing utilizes two or more processors connected by a communication network, and, each of these processors may work simultaneously on non data-dependent sub-functions of the algorithms to achieve faster execution. A parallel architecture may consist of a number of processors, and each of these may execute sequentially under program control or be totally data driven. In either case, the ultimate goal is to achieve a real time response by assigning concurrently executable instructions to different processors.

As the cost of hardware is reducing day-by-day, parallel processing architectures with large numbers of processing elements and various communication networks are becoming viable. These architectures have sufficient processing power to execute programs for real time systems. Therefore, proper utilization of these resources, and intelligent division of the work load between processors could achieve extremely fast algorithm executions. This is precisely the thesis of this research.

Any program, or task, may be represented by a program graph. As shown in figure 1, a program graph consists of nodes representing instructions, and connection arcs for the data-dependency between nodes. The parent nodes of any node are those which supply data to it. Similarly, dependents of a node are those to which it supplies data. A node is ready for execution only after its parents have completed execution, and their results have reached the node. The time expended in executing the instruction represented by a node, is the execution duration of the node, while time expended in communicating the result to the node's dependents, is the communication time. Data is passed between nodes in the form of tokens. Multiple tokens may be required to represent the result produced by any node. Instructions represented by nodes in a program graph determine its granularity, and this increases with the complexity of the nodes. When nodes represent entire functions, the graph is said to be coarsely granular. On the other extreme, if individual nodes represent single instructions, the graph is finely granular. For the proper utilization of hardware resources to get the fastest execution; granularity, parallelism, and communication, must all be considered.

Conversion of a source program written in an application language, to a program graph, is done by a compiler. Provided the problem is similarly coded, the graph generated by the compilation of different languages would be similar for a given architecture. This is analogous to the situation where compiled codes of a problem written in different high-level languages are the same for a given uniprocessor.

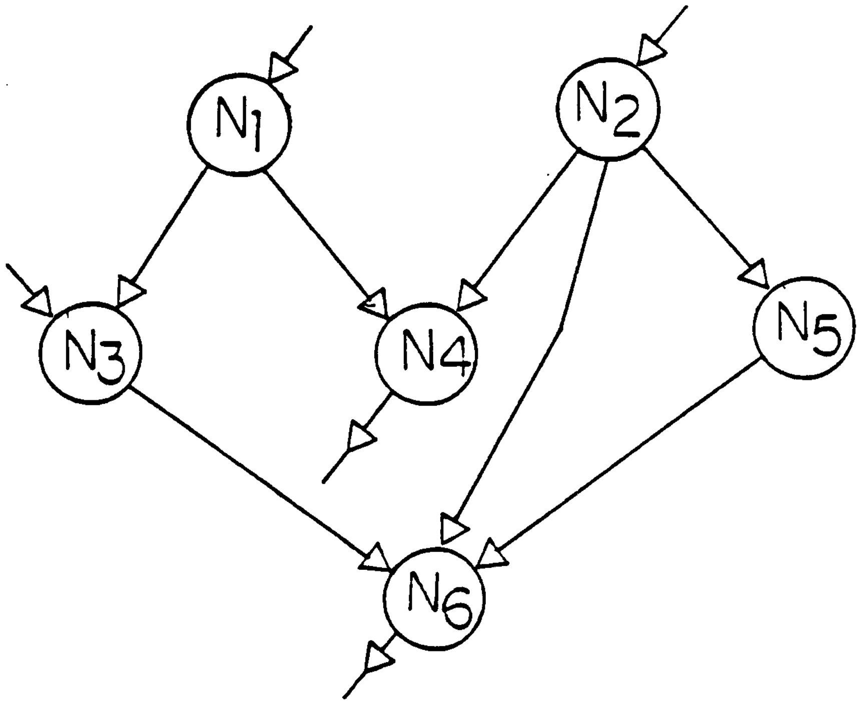

Figure 1. A program graph

In the case of a uniprocessor system, all instructions are allocated to it. However, in a parallel processing environment, the instructions have to be allocated judiciously to all the available processors for parallel execution, so that the application program is executed within the minimum amount of time. This important function of partitioning a program graph and allocation of its nodes to different processors, with a view of obtaining fast execution, is done by an allocation algorithm.

The total execution duration of any program graph, in a parallel processing environment, can be subdivided into the actual execution duration of nodes, and the time spent in internode communication. It is the function of the algorithm to minimize the sum of these two durations. In order to minimize the actual execution duration of the nodes, they are to be allocated to different processors, so that executable nodes do not have to wait for the allocated processor to complete its previous execution. This, however, results in increased internode communication duration, as only communication between nodes allocated to the same processor involves zero time duration. Minimization of internode communication duration requires the allocation of all nodes to a single processor, since interprocessor communication involves a finite time as opposed to intraprocessor communication which does not expend any time. This extreme, however, maximizes queueing delay, thereby maximizing the actual time duration of execution of nodes.

A balance therefore, has to be drawn between minimization of queueing and interprocessor communication delays. Thus, only an optimal amount of parallelism has to be exploited by the algorithm. Factors such as execution speed of the hardware, bandwidth of the network and granularity of the graph, all affect the amount of parallelism that should be exploited for an optimal allocation.

An allocation algorithm may either be static or dynamic, depending upon whether it makes allocations prior to, or during run time of the application program. Both methods have their own advantages and disadvantages. A properly designed dynamic allocator may perform better, especially in problems involving inputs from the external world. Since the allocator uses dynamically updated global information of all processors, a more efficient allocation is made. However, communication of the status of individual processors to the allocator may take a large share of the network bandwidth. If sufficient information is not dynamically supplied to such an allocator, it may lead to a bottleneck. The advantage of a static allocation algorithm is that it does not contribute to the actual run time of the application program. Well designed static allocators may be nearly optimal. It should be noted that the problem of allocation is NP-complete.

The goal of any allocation algorithm is to minimize run time of the application program. This is attained by minimization of the actual time-of-execution of nodes, interprocessor communication time and run time analysis of the graph.

1.2 LITERATURE REVIEW

This research includes a discussion of some algorithms that have been developed by researchers, in an effort to find a near optimal solution to the NP-complete problem of allocation. Some of these algorithms have been directed at specific architectures and applications. Although, the heuristics on the basis of which these algorithms have been developed are different, all of them aim at the realization of real time response.

Hong, Payne and Ferguson [1] have developed an algorithm for dynamic hypercube architectures. Their scheme divides the original program graph into disjoint tree shaped partitions. Nodes of some of these trees have only one input, while those of others have only one output. Individual trees are then mapped onto distinct faces of the hypercube. Every node in a tree is allocated to a distinct processor, located as close as possible to the processor which computes its parent.

Campbell [2] proposed a static allocation algorithm for 3-D bussed cube architectures. His algorithm consists of local and global allocators. After topologically sorting the nodes in a breadth-first manner, the local allocator assigns them to individual processing elements of the Hughes dataflow machine. In selecting a suitable processor for any node, the allocator applies two heuristic cost functions, namely, parallel processing cost and communication cost; and allocates the node to the processor returning the lowest total cost. The global allocator is similar to the local allocator, but only works on larger parallel modules consisting of several nodes, and allocates each of these modules to different sections of the hardware. Due to the amount of computation involved in allocating each node, this algorithm takes a long time to arrive at an optimal result.

Ho and Irani [3] proposed a static allocation algorithm which simulates runtime environment. Their scheme requires much information about the changing environment. For any node, this algorithm selects the processor which gives the best performance in simulation. As this algorithm also involves extensive computation, it is a lengthy process.

For graphs which are not so finely parallel, a lot of scheduling algorithms have been developed. While some of these algorithms totally neglect interprocessor communication time, others try to avoid it by redundant execution of nodes on various processors. Markenscoff and Liaw [4] have developed allocation schemes for distributed systems. Their schemes, namely, branch and bound, greedy and local search, do not allow interprocessor communication. Hence, nodes upon which any node is dependent for its input data, also have to be allocated to the same processor. This results in redundant execution of many nodes. The tradeoff in this case is between redundancy in execution and interprocessor communication time.

CHAPTER TWO

ALGORITHM

2.1 INTRODUCTION

As mentioned in the earlier chapter, an allocation algorithm is crucial to the exploitation of parallelism in a program graph and the proper utilization of hardware resources. An efficient, optimal map of nodes to processing elements, is obtained only when parallelism and sequentialism in the program graph, execution durations of the various instructions, and interprocessor communication durations are all viewed in the proper perspective. Only an algorithm which considers all these important factors arrives at a practical map, and makes a realistic estimate of the duration of execution.

Time taken in interprocessor communication has not been given sufficient importance in some of the allocation schemes described earlier. Communication between processors involves a certain finite time due to transmission of data tokens. Only in graphs whose nodes are of coarse granularity, wherein the execution time of any node is much larger than the commmunication time, can the latter be totally neglected. The significance of interprocessor communication time delay increases as its ratio with execution time increases. This is of prime importance when communication time is comparable with execution time. Due to this communication delay, even an optimally allocated graph may not execute within the minimum time as dictated by its critical path (to be discussed in the following section). Hence, interprocessor communication time has to be given due importance while allocating a program graph. It may be necessary to sacrifice a certain amount of parallelism due to the interprocessor communication involved.

2.2 CRITERIA

Certain factors are important to an allocation algorithm. These are earliest time, latest time, critical paths, network characteristics, processor execution and communication characteristics, etc.

Nodes in a program graph can be executed only when they have received their individual inputs. Thus, the earliest that a node is executable is when all of its parents have completed execution and the results have been passed on to the node. A term called 'earliest time of the node' follows from this. The earliest time of a node specifies the earliest that the node may begin execution, and is determined by the input that arrives last (deterministic input). It depends upon execution time durations of previous nodes, and is defined as being equal to the time at which all inputs to the node are available, provided that all the previous nodes have executed at their respective earliest times. It may also be defined as the maximum of the sum of earliest times of parent nodes and their respective execution time durations. The earliest time of a node with no parents is equal to zero. The algorithm represented in appendix B uses the latter definition to determine earliest times of nodes in a program graph.

When all nodes in a program graph execute at their earliest times, the graph is executed in the minimum possible time duration. This time duration is called the 'minimum time of execution.' It is not possible to execute a graph within a time frame less than the minimum time of execution. An allocation that achieves execution in the minimum time of execution is the optimal one. Thus, the minimum time of execution gives a theoretical lower bound for the execution duration of the graph, and is a good measure for evaluating the performance of an allocation algorithm. It is to be mentioned here that the definition of minimum time of execution assumes zero communication delay between nodes. Hence, in practical systems, in which interprocessor communication has a finite significant value, it is not possible to make an allocation that will execute the graph within this time frame.

The earliest time of a node, which was defined earlier, is the time at which all the inputs to the node are available, provided parent nodes had executed at their respective earliest times. Thus, the earliest time of any node is determined by the input that arrives last. The other inputs may arrive at any time prior to the arrival of the deterministic input. Those parents which are the sources of these non-deterministic inputs may execute at a time later than their earliest times, and the program graph could still execute within the minimum time of execution. This leads to the definition of another term called the 'latest time'. The latest time of a node is the latest it may execute, without prolonging the execution of a graph, beyond its minimum time of execution. It is hence obvious that the latest time of a node with no dependents, is equal to the difference between the minimum time of execution of the graph and the execution duration of the node. The latest time of any other node is equal to the difference between the minimum of the latest times of its dependents and its execution duration. This definition is implemented in the algorithm shown in appendix C.

A node may begin execution any time between its earliest and latest times without increasing the total execution time of the graph. This time period, between the node's earliest time and latest time, is known as the 'slack' or the 'optimally enabled state' of the node. It is highly desirable that every node should begin execution in its slack to achieve minimum time duration of execution of a program graph.

The earliest time and latest time of certain nodes are found to be equal. These nodes are called critical nodes, as delaying their execution beyond their earliest times results in an increase in the duration of execution of a graph by an amount equal to the delay. In order to achieve minimum duration of execution, it is very important that these critical nodes be scheduled for execution at their respective earliest times. Non-critical nodes may be scheduled for execution, any time between their respective earliest and latest times, without sacrificing the speed of execution.

Study of many program graphs has revealed that critical nodes occur in sequence thereby forming a critical path that runs through the program graph. A graph may have one or more such critical paths. Nodes in a critical path are not only critical but also sequential.

Communication is very important in a parallel processing environment. To capitalize on gains made by parallel processing, an efficient communication network is necessary. The allocation algorithm should schedule the flow of tokens so that network contentions do not occur. The characteristics of different networks vary. A star network can be implemented with minimum hardware, however, it allows only one pair of processors connected to it to communicate at a time. The communication distance between any two pairs connected to a star is a constant. A crossbar network between the processors, would be totally non-blocking, allowing simultaneous communication between processors. Even in this case, a processor may not receive information simultaneously from two or more processors. Study of program graphs has revealed that although some nodes within each module can be executed in parallel, communication between nodes can be scheduled sequentially without a great loss of performance. Thus, the allocation algorithm has to schedule transmission of data from one node to another so that network or processor contentions do not occur, and data reaches destination nodes within their slack times.

The next factor to be considered is the communication characteristic of individual processors. Usually, processors can either send or receive only one data token at any given time. As mentioned before, many nodes are to be assigned to each processor. Each node assigned to a processor, may send or receive one or more tokens to or from other processors. The allocation algorithm should schedule all communication of this processor so that contention does not occur. Particular attention is to be paid in scheduling communication caused by directly data dependent nodes that are assigned to the same processor. A parent node assigned to a processor will need to communicate its result to all its dependents, while one of its dependents which is assigned to the same processor, may simultaneously need to get its data from its other parents. This situation is illustrated in figure 2. Therefore, the order in which results have to be sent to dependents, and the instants at which these have to be done, is to be intelligently decided by the algorithm.

Due to the limited instruction storage capacity of each processor, only a certain number of nodes can be stored at any given time. Likewise, the number of nodes that the architecture can hold at any given time is also limited. If the number of nodes in a program graph is greater than the storage capacity of the architecture, or even if more nodes than can be stored in a processor has to be allocated to it, some nodes will have to be loaded during run time. This can be done in two different ways. In the first method, the allocation algorithm divides the node-to-processor map into many modules, and successively loads these

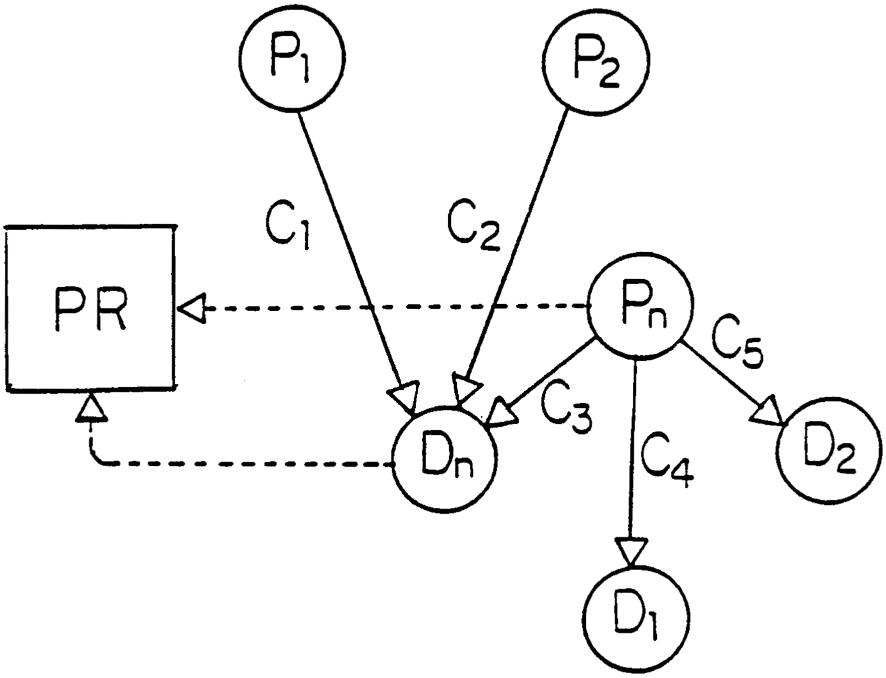

Figure 2. Illustration of communication

modules as each completes execution. Therefore, after execution of each module, a certain amount of time is expended in loading the nodes of the next module on to the processors. This method is especially suited to architectures whose communication networks have small bandwidths. In architectures having more extensive communication structures, a different method may be used. As and when nodes are executed in the processors, they are removed from the instruction store, thus creating free memory space in which new instructions may be loaded.

New nodes may therefore be loaded on to the processors by the allocation algorithm, simultaneously with the execution of the program; and for doing this a load time has to be associated with each node. It is at this load time, that the allocation algorithm loads the node on to the processor. This method could be well implemented in a non-blocking communication network architecture, such as a crossbar, and also in blocking networks when the program graph is coarsely granular.

Allocation of critical nodes, and the exact scheduling of their execution, is extremely crucial to the execution of a program graph, in the minimum possible time. Similarly, scheduling the execution of non-critical nodes within their slack times is also important. It may not always be possible to do this due to communication characteristics of the architecture, and fineness in the granularity of a program graph. The exact scheduling of critical nodes is attempted by assigning all the nodes in a critical path to a single processor. In addition to doing this, the inputs to these critical nodes should reach the assigned processor prior to their critical times. In allocating non-critical nodes, the algorithm should locate a processor that is idle during the node's slack period, and allocate the node to it if all inputs to the node can reach it within its latest time. If one such processor is not available, the algorithm should allocate the node to any other processor, so that delay in execution of the node beyond its slack time is a minimum.

2.3 BRIEF OUTLINE

The algorithm adopted in this research, involves the determination of critical paths, and allocation of each critical path to an individual processor. Once the processors for all critical paths are identified, the algorithm then sorts all the nodes in an increasing order of their latest times. Nodes with the same latest times are sorted in a decreasing order of their earliest times, which is the same as the increasing order of their slack times. Nodes in a program graph are allocated one-by-one as per the order established earlier, on the basis of certain principles that are described in a later section. The various phases in the algorithm are listed below.

- Determination of earliest times.
- Determination of latest times.
- Determination of critical paths.
- Ordering the nodes for allocation.
- Assigning of critical paths to processors.
- Allocation of nodes to processors, scheduling of their execution and their communication.

The first four, of the above phases, fall into the pre-allocation processing group, while the last two compose the allocation group.

2.4 ASSUMPTIONS

Assumptions made in the design of this allocation algorithm are described in this section.

Very large programs are already partitioned, if possible, into smaller modules that have very little, or no data-dependency between them. This may be done by looking at the syntax of the high-level language which is the source of the program graph. The algorithm works on these modules, one at a time, and maps them on to a hardware submodule (cluster, a side of a hypercube, etc.).

The nodes in each partitioned module of a program graph are numbered by some other algorithm, so that a parent always has a numerically lesser node number than its dependents.

The algorithm assumes the availability of an unlimited number of processing elements on to which it can allocate the program graph. This is justifiable as hardware cost keeps reducing with advances in technology, and a lot of processing elements may be used in any architecture. Hence, the graph may be spread in an optimally parallel fashion without any loss of parallelism due to hardware constraints.

The algorithm needs as its input, and hence assumes the availability of time durations of execution of various instructions that are associated with the nodes in a program graph. It is possible to determine the execution duration of all the instructions of a processor.

Another input needed by the algorithm, is a list of communication time durations for a single token between various processors in an architecture. Interprocessor communication time durations depend upon the structure of the interconnecting network between processors in a module of the hardware architecture.

Communication between processors is by token passing. The outputs produced by different nodes may be of varying lengths, and hence require different numbers of tokens.

2.5 INPUTS

The inputs to the allocation algorithm are listed below.

- Table of execution durations of the various instructions.

- Table of interprocessor communication times for a single token.
- Program graph, with the nodes numbered in an order of dependency.
- Number of tokens needed to represent the output of each node.

2.6 DETAILED DESCRIPTION

As mentioned earlier, the algorithm works in various phases that fall into two distinct groups. The analysis group comprises of the determination of earliest times, latest times and slack periods of nodes, determination of critical nodes and critical paths, and the ordering of nodes for allocation. The allocation group involves the allocation of critical paths, scheduling execution of nodes and scheduling of interprocessor communication.

The earliest times of all nodes are determined by using the algorithm shown in appendix B. Determination of earliest times begins with node number one, the root node with no parents. Finally, the minimum time of execution of the program graph is determined.

The next phase in the analysis of a program graph, is the determination of latest times and slack durations of all nodes. The algorithm for this is shown in appendix C. Determination of latest times begins with the final node having no dependents and the largest node number. The latest time of any node is determined by the least value of the latest times of its dependents.

Critical nodes and critical paths are next identified. All nodes having their earliest time equal to their latest time are critical nodes. These nodes have no slack durations, and have to be scheduled for execution exactly at their critical times to avoid lengthening program execution. Critical nodes occur sequentially in a program graph forming one or more critical paths.

Identification of critical paths is done as follows. Any critical node having no inputs from other critical nodes is considered to head a critical path. If only one parent of a critical node, is critical, the node is identified as belonging to the critical path formed by this parent. In certain cases, two or more parents of a critical node are also critical, and on the basis of certain principles, the critical node has to be identified as belonging to one of the critical paths formed by its parents. Figure 3 illustrates such a case, where both parents of the critical node N are critical, and only one of these parents is the last node of a critical path (critical path 1). The other parent, in critical path 2, already has a critical successor identified with it. In such cases, the node is identified with critical path 1.

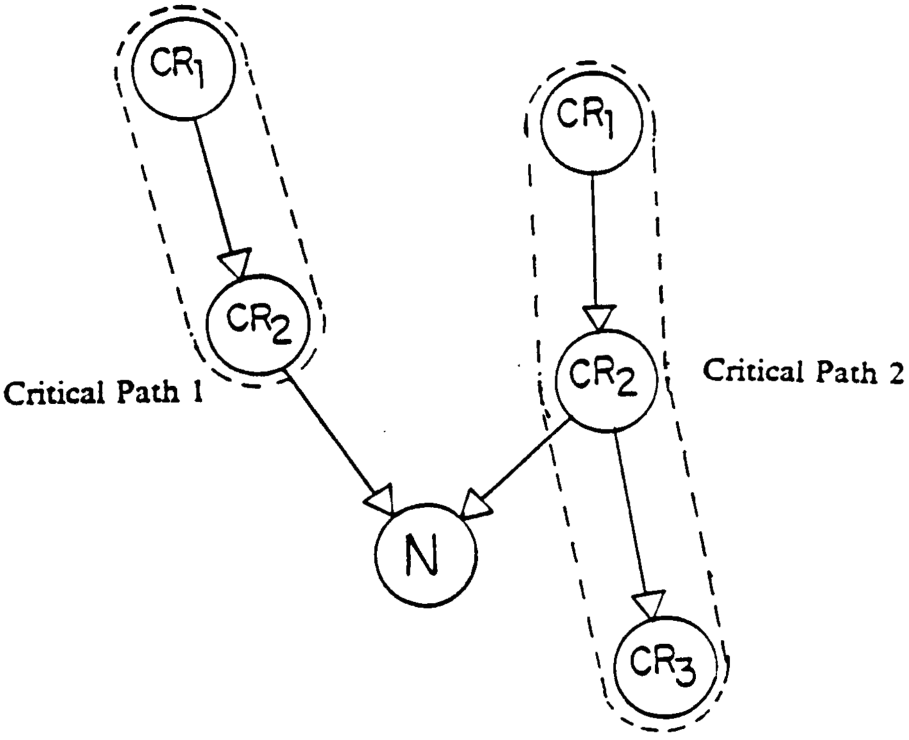

Figure 3. Identification of critical paths (case 1)

In another case, as illustrated by figure 4, both critical parents of the critical node N, are the last nodes in their respective critical paths. In such cases, the node is identified as belonging to the path formed by the parent that completes execution last, and hence determines the critical time of the node. If both parents complete execution at the same time, the node is identified with the path formed by the parent that generates more output data, so that a smaller amount of time is lost in interprocessor communication.

The final phase in the pre-allocation analysis group is the ordering of nodes for allocation. Nodes with lower values of latest times have to be allocated first. Amongst nodes with equal latest times, those with shorter slacks are given priority. To do this, nodes are numbered in an increasing order of their latest times, and those with equal latest times are numbered in a decreasing order of their earliest times.

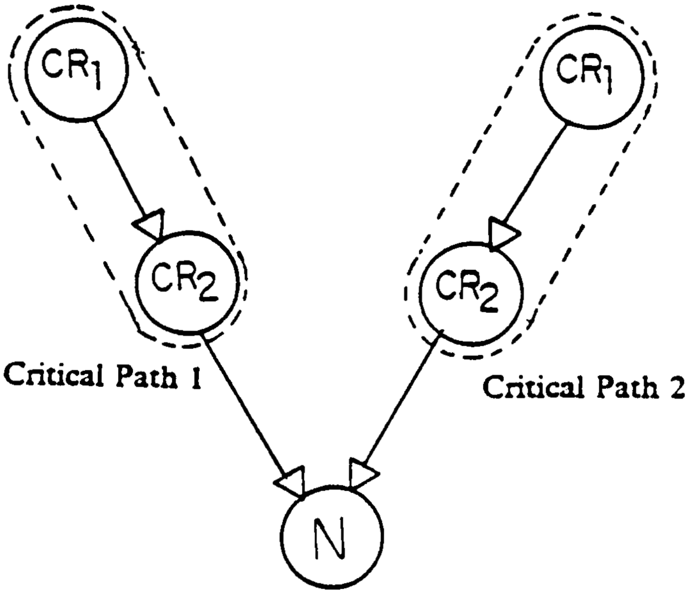

Figure 4. Identification of critical paths (case 2)

Critical paths determined earlier are each identified with a processor. The algorithm considers one critical path at a time, and identifies nodes in the path with a single processor free to execute them at their critical times. These nodes are later allocated to the identified processor. As the algorithm assumes the availability of an unlimited number of processing elements, it can be guaranteed that an idle processor will be found for allocation of each critical path. The reason for allocating all nodes in a critical path to a single processor, is they are essentially sequential, and the critical time of any node is exactly at the completion of execution of its parent node in the critical path. Allocation of such data-dependent nodes to different processors would definitely lead to interprocessor communication delay. If on the other hand, these nodes are allocated to the same processor, communication delay is introduced only if data tokens coming from other parents cannot reach the processor within the latest times of the nodes.

Both critical and non-critical nodes are next allocated sequentially in the order established earlier. Allocation of non-critical nodes involves the determination of a processor that is idle to execute the node during its slack time, and is also free to communicate with the processors that have the node's parents and dependents assigned to them.

Determination of a processor for non-critical nodes is carried out by the application of certain principles that are discussed next. It is apparent that any node, and any of its parents, cannot be executed in parallel due to the data dependency between them. The relationship between them is essentially sequential, and the node maybe executed only after the execution of its parents. The parents of a node may however, be executed in parallel and can be allocated to different processors. In order to avoid delay due to interprocessor communication, the node has to be allocated to the same processor as its parent.

Figure 5 shows a node N with earliest time E_n and latest time L_n. Ideally, execution should begin at a time between E_n and L_n. In most cases a processor, to which one of the parents P_a, P_b or P_c is assigned, will be free to execute node N between E_n and L_n. The node N is then allocated to this processor. In case two or more processors which have parents assigned to them, are free to execute node N some time between E_n and L_n, an intelligent selection has to be made. It is assumed that t_a, t_b and t_c are the beginning times of execution, and d_a, d_b and d_c, are the durations of execution and a, b and c are the number of tokens in the result of parent nodes P_a, P_b and P_c respectively. If either a, b or c is unusually large (result being large, such as a string), then node N is allocated to corresponding processors PR_a, PR_b or PR_c respectively. This eliminates excessive communication, and hence reduces network or processor contentions. If a, b and c are all comparable, then node N is allocated to the parent processor (processor to which one of the parent nodes is allocated) that returns the highest value for the expression given below.

$$tx + dx + x*k \qquad (1)$$

where x = a, b, c and k = time duration of communication of one token.

In the event that none of the parent processors PR_a, PR_b, PR_c etc. are free to execute N in its slack period, any processor (including parent processors) that returns the minimum value for the expression given below is selected.

$$\text{Executable time} = \text{MAX}(X,Y) \qquad (2)$$

where X is the time at which all tokens arrive from the parent processors, and Y is the earliest the processor is free to execute the node.

The assumption of the availability of an unlimited number of processors, assures that a free non-parent processor can be found to execute the node at any time. If the time at which data can be sent to this processor, from the node's parents, is less than executable times of the parent processors, the node is allocated to it; else it is allocated to the parent that is free earlier. It then incurs queueing delay. Hence, a tradeoff is made between queueing delay and communication delay.

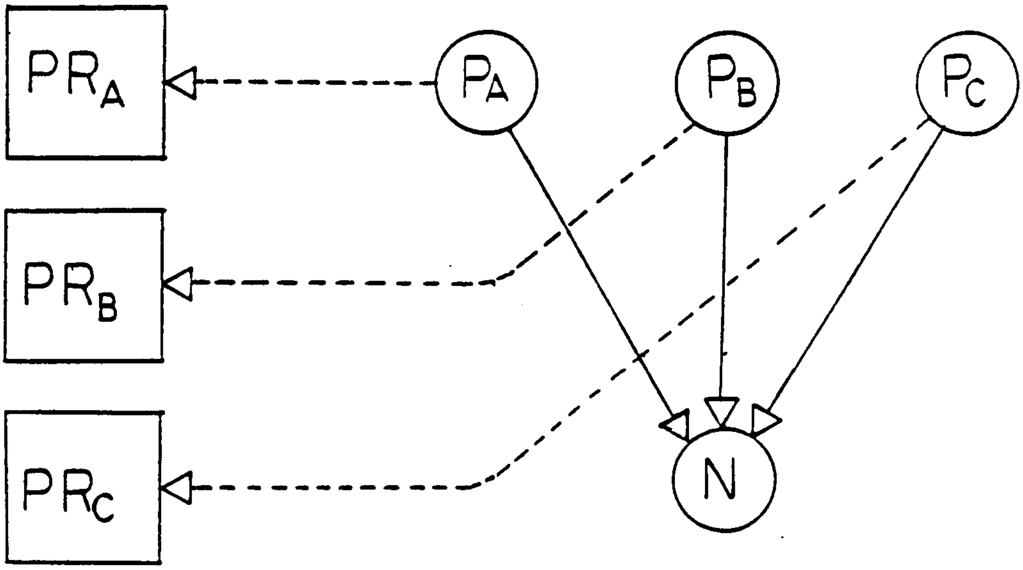

Figure 5. Allocation of node 'N' to a parent processor

Figure 6 shows an application of the principle mentioned earlier. The node N is allocated to PR_b so that communication between node P_b and N does not have any effect. The time t_n at which the node may begin execution is when its other inputs from P_a and P_c have reached P_b.

As a processor can receive information from only one processor at a time, communications from P_a and P_c to PR_b, have to be done sequentially. This is illustrated in figure 7. In this case, t_n is the exact time at which node N can execute. If t_n lies between E_n and L_n, or if E_n and L_n are after t_n, then the node N may be executed without any delay, else an inevitable communication delay is introduced.

In figure 8, parents P_1 and P_2 of N_2, have been allocated to processors PR_1 and PR_2 respectively. The dependents of P_1 are N_1, N_2 and N_3, and those of P_2 are N_2 and N_4. The allocation of node N_2 is now explained. Supposing the order established in pre-allocation mode returned a higher priority (lower number) for N_2, than for N_1 and N_4, N_2 could be allocated to either PR_1 or PR_2, depending upon which one of them returned a higher value for expression 1. If one of the parent nodes generates comparatively a large amount of data, N_2 is allocated to that parent's processor. On the other hand, if N_2 is given a lower priority than N_1 and N_4, it cannot be allocated for immediate execution on either PR_1 or PR_2, due to N_1 and N_4 already being allocated to them. In such an event, the processor that returns a minimum value for expression 2 is selected.

Once a node has been allocated to a processor, communication of outputs from the node have to be scheduled. Tokens have to be sent to the dependents in an increasing order of their latest times. When two or more dependents have the same latest time, output is first sent to the node with a greater number of dependents. In figure 9, output is first scheduled to node D_a.

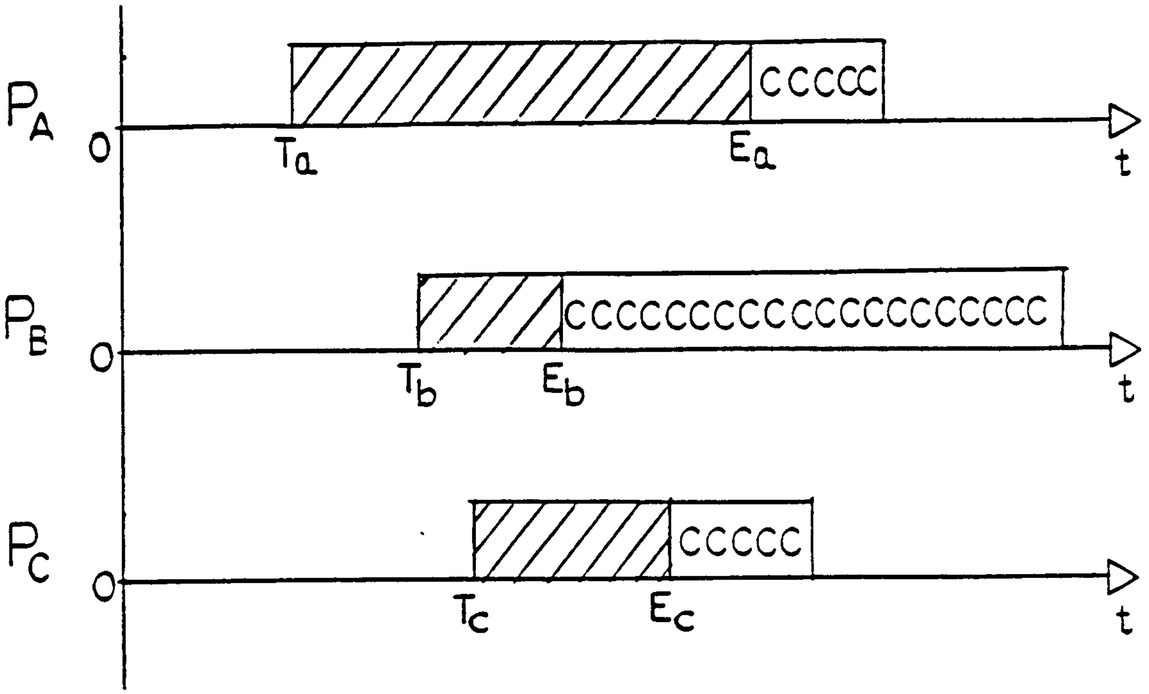

Figure 6. Execution and communication durations of parent nodes

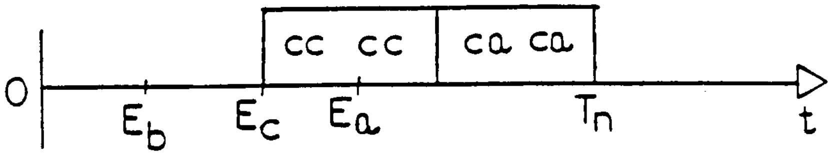

Figure 7. Illustration of earliest executable time

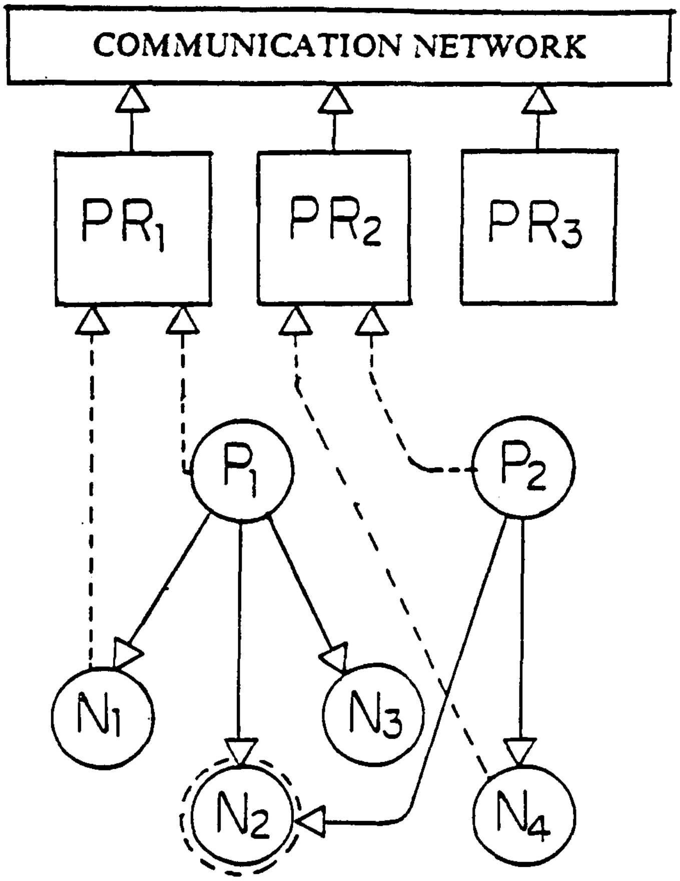

Figure 8. Illustration of allocation

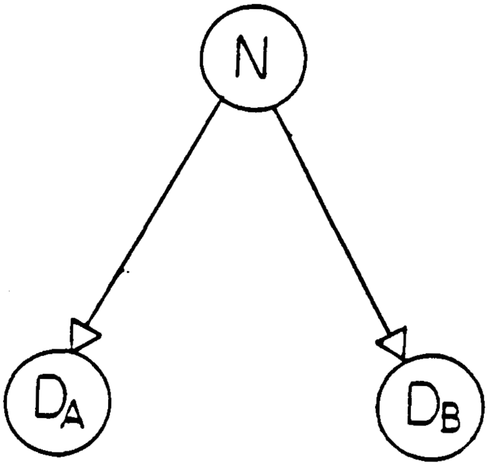

Figure 9. Illustration of output priorities

Another criterion that influences communication scheduling, is the earliest time of dependent nodes. Although it is desirable to have the output routed to dependent nodes just before their earliest times, doing it much earlier to this is not advantageous. In figure 10, although $l_a < l_b$, output from node N may be first routed to dependent D_b, if the output to dependent D_a can still be routed so that it reaches D_a within its slack time.

When one or more dependent nodes are critical nodes, the output should first be directed to these nodes without delay. A little delay, even beyond the latest time, can be tolerated by non-critical nodes.

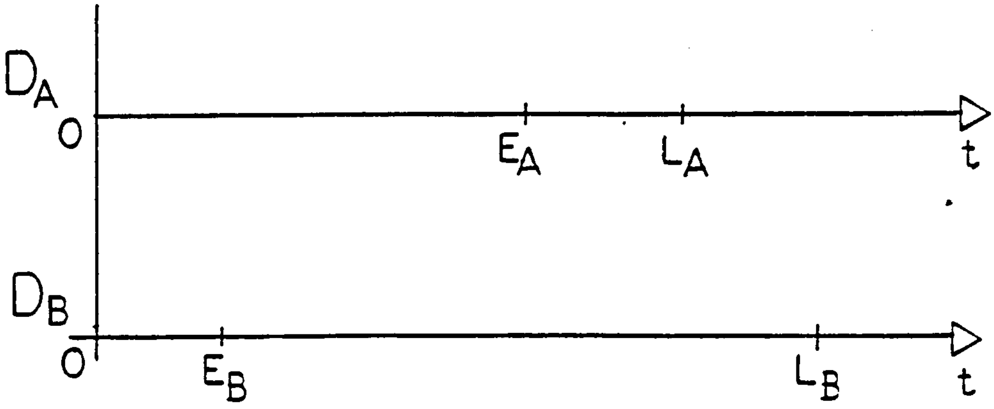

Figure 10. Slack durations of nodes

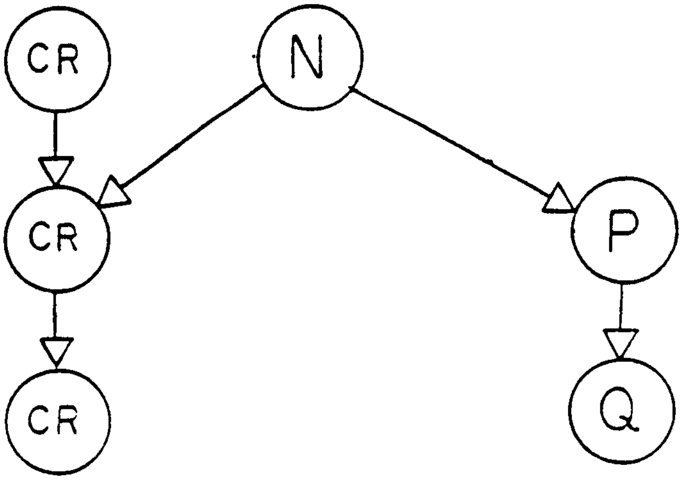

Figure 11. Illustration of input to critical nodes

Every time communication is scheduled between two processors due to token passing between nodes allocated to them, it is recorded, so that other communications are not scheduled to the processors at those same time. Therefore, communication durations of all processors, due to data transfer between nodes allocated to them have to be recorded. In blocking communication networks, even various communication durations of the network have to be recorded. Likewise, to avoid scheduling two or more nodes for execution simultaneously on the same processor, execution duration (busy times) of each processor is recorded.

The priorities in which outputs of a node are to be sent to its dependents are established immediately after its allocation. However, actual scheduling of these communications can be made only after the dependent nodes are allocated. Thus, when a node is allocated, its inputs are actually scheduled, while its outputs are given priorities. Scheduling of inputs can be done only after all of its cousins (other dependents of its parents) have also been allocated.

In figure 12, P_a and P_b are the parents of node N, and nodes C_1, C_2 and C_3 are its cousins. The dependents of P_a are N and C_1, and the output from P_a is prioritized as (N, C_1). The dependents of P_b, and its output priorities are given by (C_2, N, C_3). In this case, the input to N from P_a can be scheduled immediately after its allocation, while its input from P_b has to be scheduled only after the allocation of C_2.

The algorithm associates certain information with each node. The following information regarding each node is supplied as input to the algorithm. Each node is given a 'node number' for identification. The 'instruction number' associated with it identifies the instruction that the node executes. The 'number of parents', their respective node numbers, 'number of dependents' and their respective node numbers, give the data-dependencies of the node. The 'number of tokens' needed to represent the output produced by the node is also given as input to the algorithm. The following information associated with each node is generated as output by the algorithm. The 'load time' specifies the time at which the node is to be sent to the processing element. The 'earliest time' and 'latest time' of the node are determined by the algorithm. The 'execution time' specifies the time at which the node is scheduled for execution on its assigned processor. The algorithm also determines the priorities and schedules for communication of results to the node's dependents.

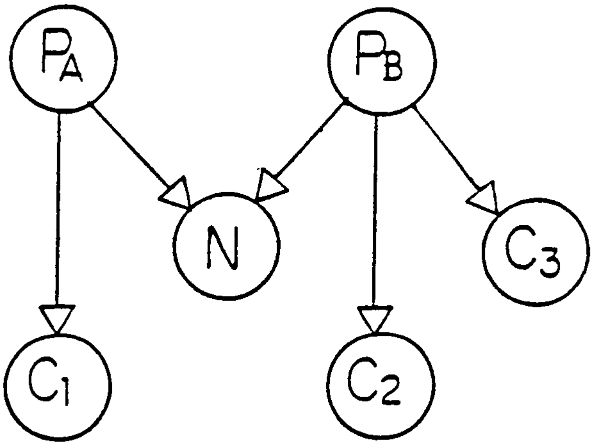

Figure 12. Illustration of input schedules

The algorithm associates some information with each processor, to aid it in its allocation. The 'number of nodes allocated' to the processor is always updated. The 'final ready time' of the processor, which is the earliest time beyond which no nodes are scheduled for execution on it, aids the algorithm in allocating subsequent nodes. The time periods during which the processor is busy executing nodes, and the time periods when the processor is communicating with other processors, are also recorded.

2.7 OUTPUTS

The outputs of the algorithm are listed below.

1. Processor allocated to each node.
2. Load-time and execution beginning time of each node.
3. Schedules of outputs to dependents, for each node.
4. Busy and communication durations of each processor.
5. Various communication durations of the network.

CHAPTER THREE

EXAMPLES

3.1 INTRODUCTION

In this chapter, two examples are presented to illustrate the allocation algorithm delineated in chapter 2. Reiterating what was mentioned earlier, it can most definitely be stated that, efficiency of an allocation made by any algorithm can be measured by the proximity between values of actual duration of execution of a graph, and the minimum time of execution as dictated by the critical paths, when time taken in interprocessor communication is assumed to be zero. The examples are presented in a form to illustrate node-by-node allocation, and scheduling of communication between nodes.

The basic time unit is assumed to be t. In the examples shown, execution duration of various instructions vary from t to 4t. Communication between nodes is by token passing. Depending upon the amount of data to be sent by any node to its dependents, multiple tokens may be necessary to represent it. For each graph, allocation has been made for different values of communication time (CT) of a single token (for CT = 0, t/2, t). To illustrate the flexibility of the algorithm with respect to the architecture, allocation has been done for both blocking and non-blocking communication networks.

When the communication time of a single token is assumed to be zero, a program graph represents a coarse grain system, in which communication time is negligible compared to execution time of individual nodes. On the other extreme, when communication time of a single token is assumed equal to t, the program graph represents a fine grain system in which the flow of information between nodes takes as much time as the execution durations of individual nodes. Thus, the allocations indicate the algorithm's applicability to different levels of parallelism.

Pre-allocation processing, which encompasses the determination of earliest and latest times of all nodes, and the ordering of these nodes for allocation, is not explained in this chapter. Only, identification of critical paths and the actual allocation of nodes are explained.

3.2 EXAMPLE 1a

The first example is shown in figure 13. In order to clearly illustrate the allocation principles, first, equal execution durations of t are assummed for all nodes. Also, with the same intention, interprocessor communication time is assumed to be zero for this example. Later examples in this chapter take the communication delay into account. Table 1 represents pre-allocation analysis of figure 13, and lists the nodes in the graph, their execution durations, earliest times, latest times, number of tokens and their order of allocation. The minimum time of execution of the graph is found to be 5t. Nodes 1, 2, 6, 7 and 9 are identified as critical nodes. When the graph is traversed from node 1 to node 9, all critical nodes are found to be sequential, thereby forming only one critical path. In this graph, there is no instance of a critical node having more than one critical parent. As per the principles of the algorithm, these critical nodes have to be allocated to the same processor, namely processor 1. Their execution times are decided after the allocation of their parents. The timing diagram shown in figure 14 represents the allocation of this graph. Nodes are allocated in the order that was established in the pre-allocation mode. Node 1 is scheduled for allocation at time '0' on processor 1. Node 2 is scheduled on processor 1 at time t, and node 6 is scheduled on the same processor at time 2t. Node 3 is to be executed between its earliest time t and latest time 2t. As processor 1, to which node 3's parent, node 1 is allocated, is busy during this time frame, node 3 is scheduled for execution at time t on processor 2. Similarly, node 4 is scheduled on processor 3 at time t. The critical time of node 7 is 3t, and its parent nodes 6, 3 and 4 have all complete execution by time 3t, hence, node 7 is scheduled for execution on processor 1 at time 3t. Node 5 is to be scheduled between its earliest time 2t and latest time 3t. As processor 1, to which its parent, node 2 is allocated, is busy during this time frame, node 5 is allocated to processor 2 and scheduled for execution at 2t. Node 8 is scheduled for execution at its earliest time 2t on processor 3 to which its parent, node 4 is also allocated. The critical node 9 is scheduled for execution on processor 1 at its critical time 4t, as all its parents have completed execution by this time.

As illustrated in figure 14, the graph completes execution by 5t, which is its minimum time of execution. Three processors have been utilized by the algorithm to schedule the graph's execution. The allocation for this case is presented in table 2, appearing at the end of this section.

Timing diagrams of figures 15 and 16, and tables 3 and 5 represent the allocation of the graph, when time duration for interprocessor communication of a single token is assumed to be t/2 and t respectively. The respective interprocessor communication schedules are shown in tables 4 and 6. All nodes are assumed to generate only one token as their outputs. Communication of tokens between nodes allocated to the same processor is assumed to involve no time. Due to the structure of the communication network (crossbar), broadcast ability in transmission of tokens to dependents, is made use of. These allocations are not explained here, as a detailed explanation of a more general case involving communication appears at a later stage in this chapter.

As illustrated in figure 15, the graph will complete execution within a time period of 5.5t when token communication duration is t/2 (half the execution duration of any node). This represents an increase of only 0.5t beyond the minimum time of execution of the graph. When token communication duration is assumed to be equal to execution duration of any node, as shown in figure 17, the graph is scheduled on three processors to execute within a time period of 7t.

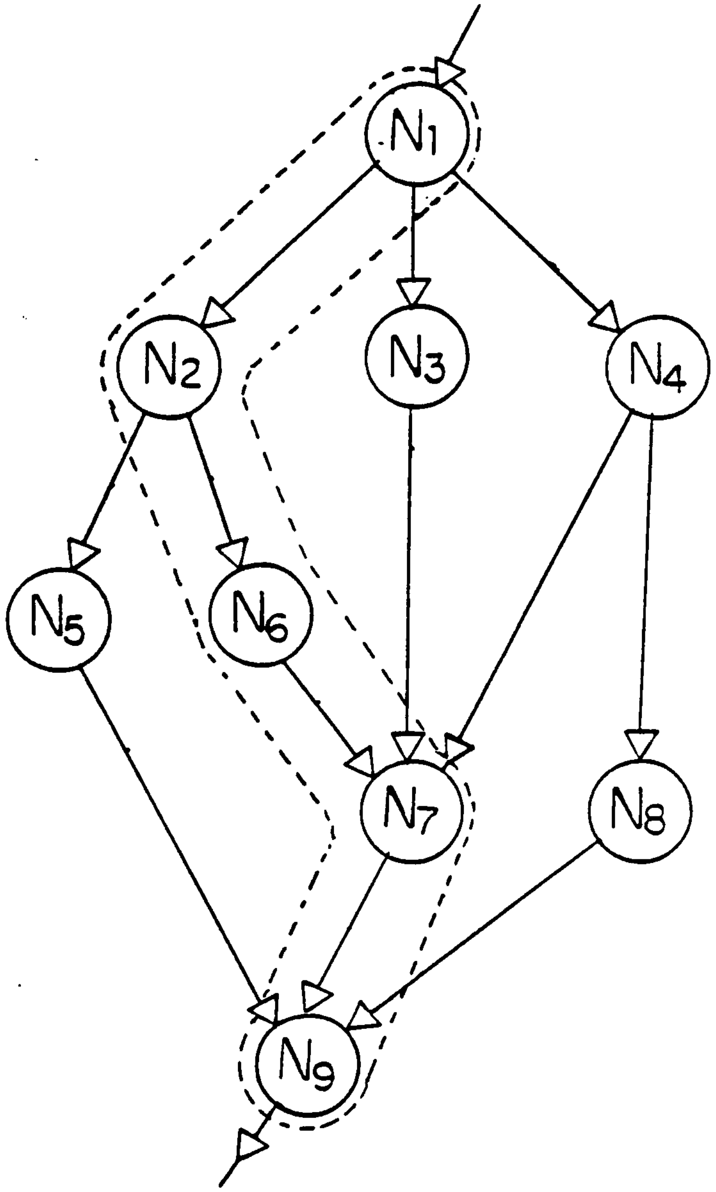

Figure 13. Program graph 1

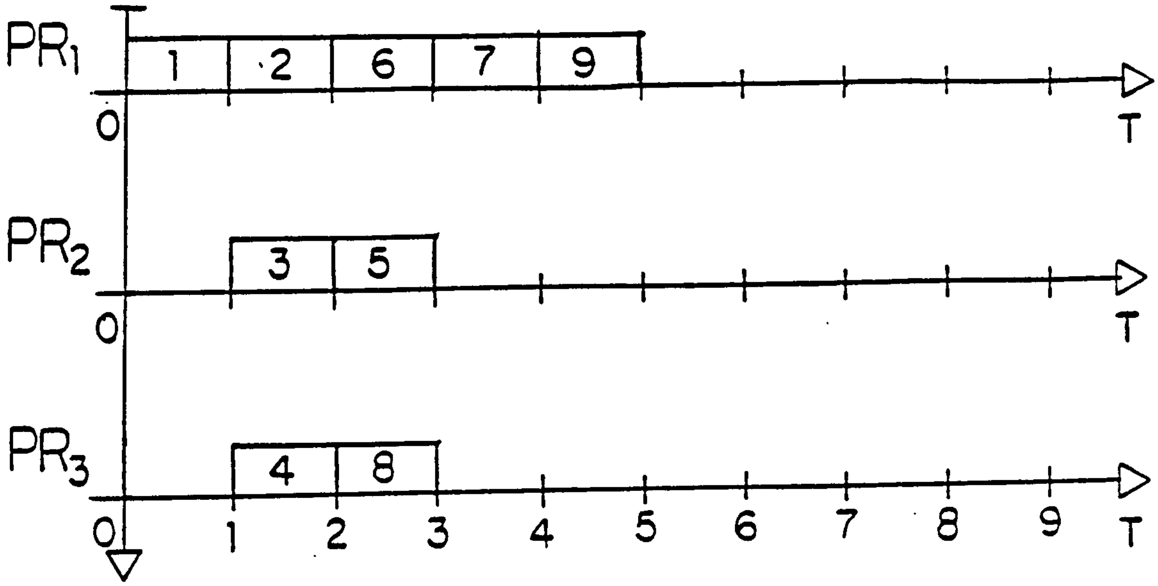

Figure 14. Timing diagram of graph represented by table 1 for CT = 0

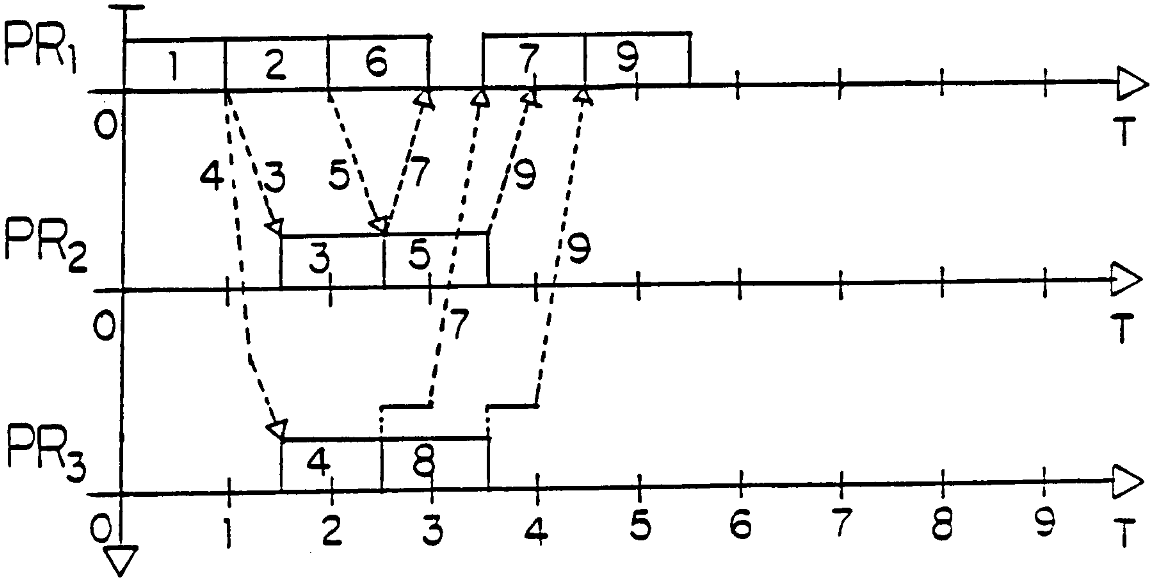

Figure 15. Timing diagram of graph represented by table 1 for CT = t/2

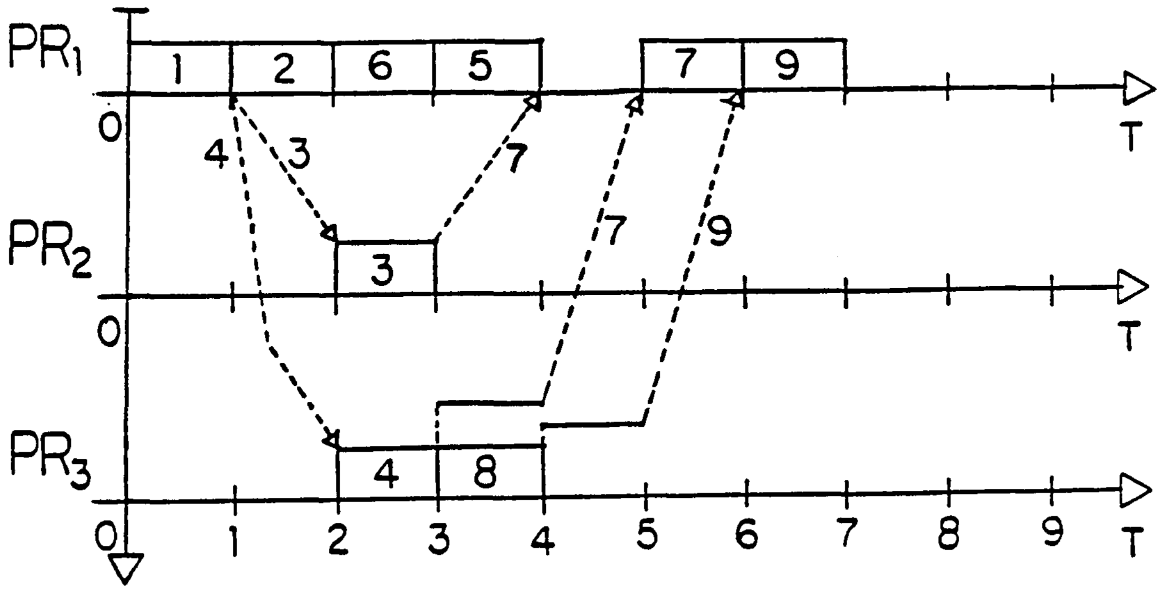

Figure 16. Timing diagram of graph represented by table 1 for CT = t

Table 1
Pre-allocation analysis with equal durations for figure 13

Node Number	Execution Duration	Number of Tokens	Earliest Time	Latest Time	Order
1	t	1	0	0	1
2	t	1	t	t	2
3	t	1	t	2t	4
4	t	1	t	2t	5
5	t	1	2t	3t	7
6	t	1	2t	2t	3
7	t	1	3t	3t	6
8	t	1	2t	3t	8
9	t	1	4t	4t	9

Table 2
Resource allocation for table 1 when CT = 0

Processor	Nodes (x,y,z)
PR1	(0,1, t) (t,2,2t) (2t,6,3t) (3t,7,4t) (4t,9,5t)
PR2	(t,3,2t) (2t,5,3t)
PR3	(t,4,2t) (2t,8,3t)
Total execution duration = 5t; Minimum time of execution = 5t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 3
Resource allocation for table 1 when CT = t/2

Processor	Nodes (x,y,z)
PR1	(0,1, t) (t,2,2.0t) (2t,6,3t) (3.5t,7,4.5t) (4.5t,9,5.5t)
PR2	(1.5t,3,2.5t) (2.5t,5,3.5t)
PR3	(1.5t,4,2.5t) (2.5t,8,3.5t)
Total execution duration = 5.5t; Minimum time of execution = 5t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 4
Schedule of interprocessor communication for table 1 when CT = t/2

Source node	Processor & exec. time	# of tokens	Schedule of communication (d,p,x,y)
N1	PR1,0,t	1	(N3,PR2,t,1.5t) (N4,PR3,t,1.5t)
N2	PR1,t,2t	1	(N5,PR2,2t,2.5t)
N3	PR2,1.5t,2.5t	1	(N7,PR1,2.5t,3t)
N4	PR3,1.5t,2.5t	1	(N7,PR1,3t,3.5t)
N5	PR2,2.5t,3.5t	1	(N9,PR1,3.5t,4t)
N6	PR1,2t,3t	1	
N7	PR1,3.5t,4.5t	1	
N8	PR3,2.5t,3.5t	1	(N9,PR1,4t,4.5t)
N9	PR1,4.5t,5.5t	1	

(d,p,x,y) : Communication to dependent 'd' allocated to processor 'p' is scheduled between 'x' and 'y'.

Table 5
Resource allocation for table 1 when CT = t

Processor	Nodes (x,y,z)
PR1	(0,1, t) (t,2,2t) (2t,6,3t) (3t,5,4t) (5t,7,6t) (6t,9,7t)
PR2	(2t,3,3t)
PR3	(2t,4,3t) (3t,8,4t)
Total execution duration = 7t; Minimum time of execution = 5t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 6
Schedule of interprocessor communication for table 1 when CT = t

Source node	Processor & exec. time	# of tokens	Schedule of communication (d,p,x,y)
N1	PR1,0,t	1	(N3,PR2,t,2t) (N4,PR3,t,2t)
N2	PR1,t,2t	1	
N3	PR2,2t,3t	1	(N7,PR1,3t,4t)
N4	PR3,2t,3t	1	(N7,PR1,4t,5t)
N5	PR1,3t,4t	1	
N6	PR1,2t,3t	1	
N7	PR1,5t,6t	1	
N8	PR3,3t,4t	1	(N9,PR1,5t,6t)
N9	PR1,6t,7t	1	

(d,p,x,y) : Communication to dependent 'd' allocated to processor 'p' is scheduled between 'x' and 'y'.

3.3 EXAMPLE 1b

Table 7, a pre-allocation analysis table, presents unequal execution durations for the nodes of the program graph in figure 13. The table also lists the number of tokens necessary to represent the result generated by each node. These are also assumed to be unequal. The earliest time, latest time and order of allocation of each node, as determined by pre-allocation analysis are also shown in it. Allocation of the graph represented by this table for token communication durations of 0, 0.5t and t are represented by figures 17, 18 and 19, and tables 8, 9 and 11 respectively. Interprocessor communication schedules for token communication durations of t/2 and t are represented by tables 10 and 12 respectively. The allocation represented by figure 18 is explained in the following paragraph.

Pre-allocation analysis returns a single critical path formed by nodes 1, 2, 6, 7 and 9. According to the principles of the algorithm, these are to be assigned to a single processor, namely processor 1. Node 1 is scheduled for execution on processor 1 at time zero. Node 2 is scheduled at time 3t on the same processor. The next node scheduled, according to the order determined, is node 4. The earliest and latest times of this node being 3t and 4t respectively, it is desirable that this be scheduled for execution as early as possible after time 3t. Processor 1, to which node 4's only parent, node 1 is allocated, is busy during its slack. Hence, node 4 is scheduled on processor 2. As node 1 needs two tokens to represent its output, it takes one time unit to communicate its output to a different processor. Hence, node 4 is scheduled for execution at time 4t on processor 2. Communication between node 1 (allocated to processor 1) and node 4 (allocated to processor 4) is scheduled to occur at 3t, and lasts for a duration of t. Critical node 6 is scheduled for execution at time 5t on processor 1. Like node 4, node 3 is also only dependent upon node 1 for its input. It is similarly allocated to processor 3, and scheduled for execution at time 4t. As the crossbar network allows broadcast communication, data transfer from node 1, to nodes 3 and 4, is scheduled to occur simultaneously. Critical node 7 is to be executed at time 6t if delay is not to be introduced in the execution of the program graph. It needs inputs from nodes 3, 4 and 6. As node 6 is also allocated to processor 1, only inputs from nodes 3 and 4 are to be scheduled. Node 3 completes execution at 5t, and its output is represented by a single token. Hence, communication between node 3 and node 7 is scheduled between time 5t and 5.5t. Node 4 completes execution at 6t, and its output is represented by two tokens, thus involving a communication duration of t. Communication between nodes 4 and 7 is scheduled between 6t and 7t. Thus, the earliest node 7 may execute due to communication involved, is 7t (its critical time is 6t). Node 7 is therefore scheduled for execution at 7t. Next in line for allocation, is node 5. The earliest and latest times of this node are 5t and 9t respectively. Its only parent, namely node 1, has been allocated to processor 1. As the duration of node 5 is t, and processor 1 is free between 6t and 7t, node 5 is scheduled for execution at 6t on processor 1. Node 8 can be executed at any time between 5t and 9t without introducing queueing delay. As node 4, the only parent of node 8, is allocated to processor 2, node 8 is scheduled for execution processor 2 at time 6t. Critical node 9 is to be scheduled on processor 1. It needs inputs from nodes 5, 7 and 8. As nodes 5 and 7 are allocated to processor 1, only its input from node 8 has to be scheduled. Node 8 completes execution on processor 2 at 7t, and produces only one token. Thus, the communication between node 8 and 9 is scheduled between 7t and 7.5t. Node 9 is scheduled for execution on processor 1 at time 11t, which is the time node 7 completes execution. Thus, the total execution duration of the graph is found to be 15t.

The minimum duration of execution of this graph, as obtained by earliest time analysis, was 14t. Allocation of the graph for zero interprocessor token communication delay results in an execution duration of 14t (the minimum value possible). Considering interprocessor token communication duration as 1/2t and t, execution durations of the allocation are found to be 15t and 17t respectively.

One observation that can be made from figures 18 and 19 is that although all the nodes in a critical path are allocated to the same processor, some of them cannot be executed at their critical times due to their data-dependency on non-critical nodes, and the finite time involved in interprocessor communication.

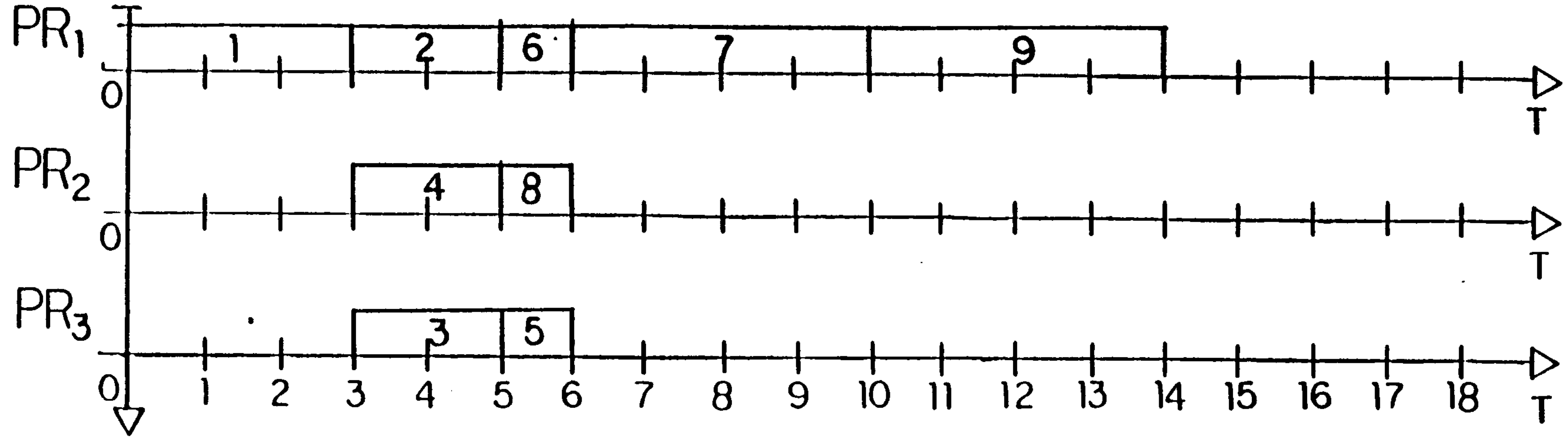

Figure 17. Timing diagram of graph represented by table 7 for CT = 0

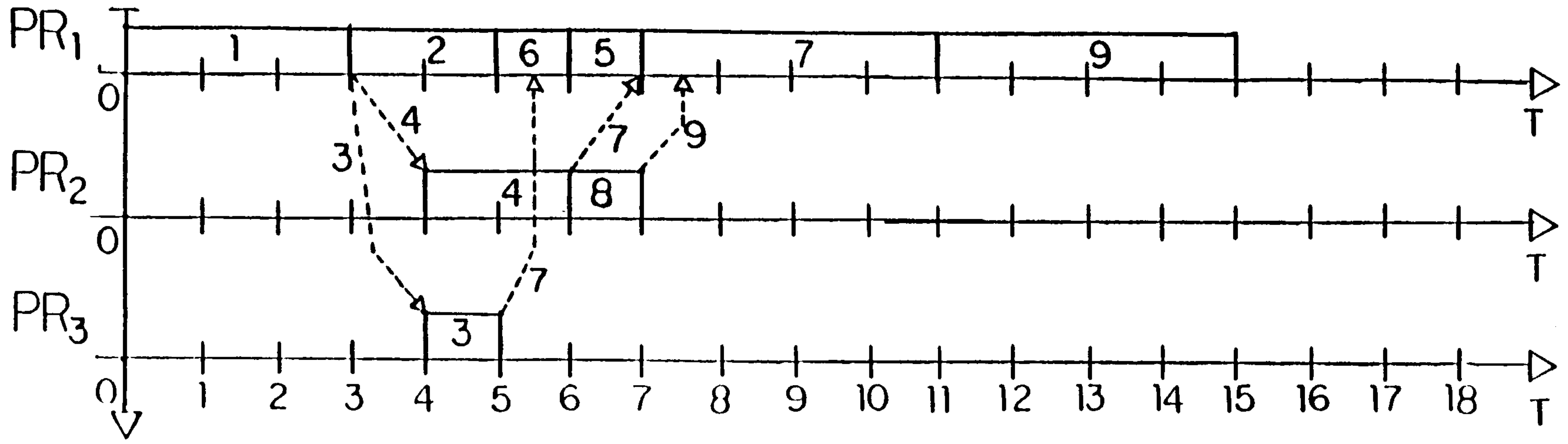

Figure 18. Timing diagram of graph represented by table 7 for CT = t/2

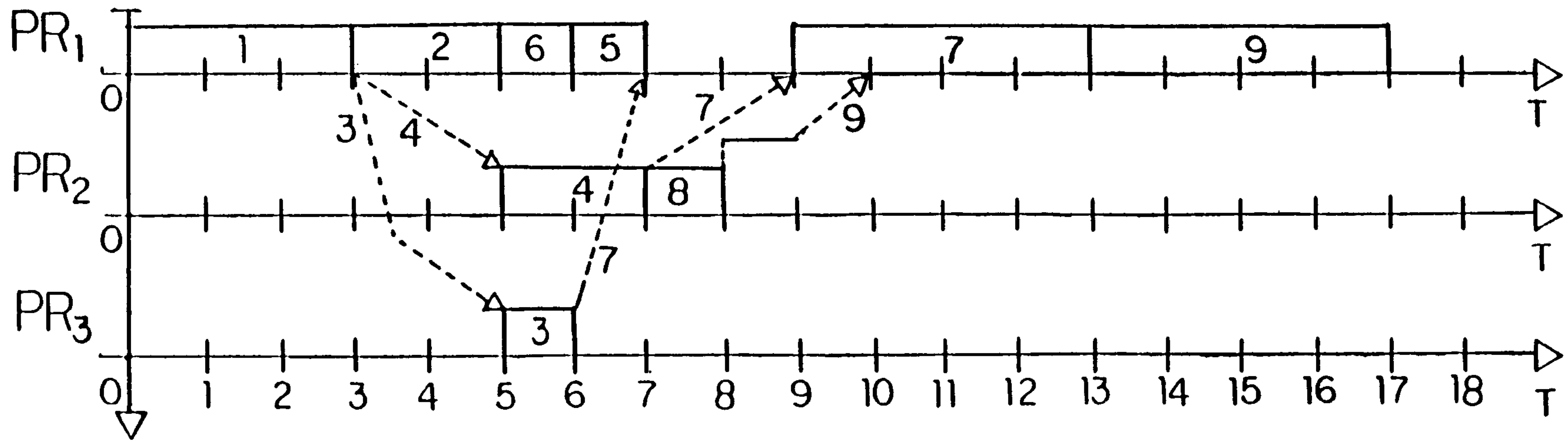

Figure 19. Timing diagram of graph represented by table 7 for CT = t

Table 7
Pre-allocation analysis with unequal durations for figure 13

Node Number	Execution Duration	Number of Tokens	Earliest Time	Latest Time	Order
1	3t	2	0	0	1
2	2t	2	3t	3t	2
3	t	1	3t	5t	5
4	2t	2	3t	4t	3
5	t	1	5t	9t	7
6	t	1	5t	5t	4
7	4t	3	6t	6t	6
8	t	1	5t	9t	8
9	4t	2	10t	10t	9

Table 8
Resource allocation for table 7 when CT = 0

Processor	Nodes (x,y,z)
PR1	(0,1,3t) (3t,2,5t) (5t,6,6t) (6t,7,10t) (10t,9,14t)
PR2	(3t,4,5t) (5t,8,6t)
PR3	(3t,3,5t) (5t,5,6t)
Total execution duration = 14t; Minimum time of execution = 14t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 9
Resource allocation for table 7 when CT = t/2

Processor	Nodes (x,y,z)
PR1	(0,1,3t) (3t,2,5t) (5t,6,6t) (6t,5,7t) (7t,7,11t) (11t,9,15t)
PR2	(4t,4,6t) (6t,8,7t)
PR3	(4t,3,5t)
Total execution duration = 15t; Minimum time of execution = 14t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 10
Schedule of interprocessor communication for table 7 when CT = t/2

Source node	Processor & exec. time	# of tokens	Schedule of communication (d,p,x,y)
N1	PR1,0,3t	2	(N3,PR3,3t,4t) (N4,PR2,3t,4t)
N2	PR1,3t,5t	2	
N3	PR3,4t,5t	1	(N7,PR1,5t,5.5t)
N4	PR2,4t,6t	2	(N7,PR1,6t,7t)
N5	PR1,6t,7t	1	
N6	PR1,5t,6t	1	
N7	PR1,7t,11t	3	
N8	PR2,6t,7t	1	(N9,PR1,7t,7.5t)
N9	PR1,11t,15t	2	

(d,p,x,y) : Communication to dependent 'd' allocated to processor 'p' is scheduled between 'x' and 'y'.

Table 11
Resource allocation for table 7 when CT = t

Processor	Nodes (x,y,z)
PR1	(0,1,3t) (3t,2,5t) (5t,6,6t) (6t,5,7t) (9t,7,13t) (13t,9,17t)
PR2	(5t,4,7t) (7t,8,8t)
PR3	(5t,3,6t)
Total execution duration = 17t; Minimum time of execution = 14t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 12
Schedule of interprocessor communication for table 7 when CT = t

Source node	Processor & exec. time	# of tokens	Schedule of communication (d,p,x,y)
N1	PR1,0,t	2	(N3,PR3,3t,5t) (N4,PR2,3t,5t)
N2	PR1,t,2t	2	
N3	PR3,5t,6t	1	(N7,PR1,6t,7t)
N4	PR2,5t,7t	2	(N7,PR1,7t,9t)
N5	PR1,6t,7t	1	
N6	PR1,5t,6t	1	
N7	PR1,9t,13t	3	
N8	PR2,7t,8t	1	(N9,PR1,9t,10t)
N9	PR1,13t,17t	1	

(d,p,x,y) : Communication to dependent 'd' allocated to processor 'p' is scheduled between 'x' and 'y'.

3.4 EXAMPLE 2

Table 13 represents the pre-allocation analysis of a more complex program graph shown in figure 20. Nodes 1, 2, 4, 5, 8, 9, 12, 13 and 14 are found to be critical. Since the identification of critical paths is not as obvious as it was in the graph of figure 13, the next few paragraphs dwell upon the determination of critical paths.

Nodes 1 and 2 are both critical, and do not have any inputs. Hence, each of these form the starting node of critical paths 1 and 2 respectively. Node 4 receives its input only from node 2, and hence belongs to critical path 2. Node 5 receives its input from both nodes 1 and 2. The critical time of node 5, which is 5t, is determined by node 1, which completes execution at 5t (node 2 completes execution at 2t). Hence, node 5 belongs to path1. Node 8 receives its input from critical nodes 1 and 4, and both these nodes complete their execution at the same time. As node 4 is last in path 2 (while node 1 is not the last node in path 1), node 8 is identified as belonging to path 2. As nodes 9 and 13 receive inputs only from nodes of path 1, both of them are identified as belonging to it. Similarly, node 12 is identified as belonging to path 2. Node 14 receives its inputs from both path 2 (node 12) and path 1 (node 13). Both these nodes complete execution at 11t. Hence, the choice of the path to which node 14 belongs, depends upon the amount of information generated by nodes 12 and 13. It is to be allocated to the path that generates more information. As node 13 generates two tokens, while node 12 generates only one, node 14 is identified as belonging to path 1.

Nodes in critical path 1 are to be allocated to processor 1, and those in critical path 2 are to be allocated to processor 2. Figures 21, 22 and 23, and tables 14, 15 and 17 represent the allocation of the graph shown in figure 20, for token communication durations of 0, t/2 and t respectively. Tables 16 and 18 represent interprocessor communication schedules for token communication durations of t/2 and t respectively. The allocation represented by figure 23 is explained in the following lines. Node 1 is scheduled for execution at processor 1 at time 0. Node 2 is scheduled processor 2 at time 0. Node 4 receives input only from node 2, and hence it is scheduled for execution at time 2t on processor 2. Node 5 receives inputs from both nodes 1 and 2. As it belongs to critical path 1, it is to be scheduled on processor 1. As the output of node 2 is represented by two data tokens, and interprocessor communication time of each token is t, communication between nodes 2 and 5 is scheduled to begin at 2t and end at 4t. Node 5 is scheduled at 5t on processor 1. Node 8 is to be allocated to processor 2. Its critical time is 5t. Although processor 2 is free at this time, node 8 cannot be scheduled for execution on it, as the input from node 1 (which completes execution at 5t) takes two time units to reach it. Hence, node 8 is scheduled to execute at 7t on processor 2, and communication between nodes 1 and 8 is scheduled to begin at 5t and end at 7t. Node 7 is scheduled for execution at time 0 on processor 3, which is the first processor that is free during the node's slack time. Node 9 receives its input from node 5 only. It is scheduled to execute at time 7t on processor 1. Node 6 receives its input from only node 2, which is allocated to processor 2. Processor 2 is free between 5t and 7t, and as the slack of node 6 is between 2t and 7t and its execution duration is 2t, it is scheduled to begin execution at 5t on processor 2. Node 12 is to be allocated to processor 2. Its critical time is 8t, and it receives inputs from nodes 7 and 8. Although node 7 finishes its execution at 3t, communication with node 12 cannot be scheduled until 7t, as processor 2 to which node 12 is to be assigned is busy communicating with processor 1. Node 7 needs two tokens to represent its output, and hence communication between node 7 (on processor 3) and node 12 (on processor 2) takes two time periods, beginning at 7t. Node 12 can be scheduled for execution on processor 2 only at time 10t, as its parent, node 8 completes execution at this time. The slack time of node 10 is between 4t and 10t. Although processor 3 is free during this time, allocation to this processor cannot be done, as it has to receive its input from node 6 allocated to processor 2. Node 6 completes execution at time 7t. However, it cannot communicate its result until 9t, as processor 2 is busy communicating with processor 3. Node 6 needs only one token to represent its output. The output can therefore reach a different processor at time 10t, which is when node 10 has to be scheduled. Both processors 1 and 3 are free between 10t and 11t, and node 10 may execute on either of them. As node 13, the dependent of node 10, is to be allocated to processor 1, node 10 is scheduled to execute on processor 1 at 10t. Node 13 is scheduled on processor 1 at 11t. By this time, both its parents, nodes 9 and 10 would have completed execution. Node 3 receives input only from node 1, and its slack time is between 5t and 10t. As processor 1, to which node 1 is assigned is busy during this time frame, node 3 can be scheduled only on processor 3 which is the next free processor. Communication between nodes 1 and 3 is scheduled to occur between 5t and 7t. Node 3 is thus scheduled to execute at 7t on processor 3. Node 14 is to be allocated to processor 1. Its parent nodes, 12 and 13 are allocated to processors 2 and 1 respectively. Node 13 finishes execution at 12t and takes two time units for interprocessor communication of its output. Node 12 finishes execution at 13t, and takes only one time unit for interprocessor communication. Hence node 14 may be allocated to either processor 1 or 2. It is scheduled to begin execution on processor 1 at 14t. Communication between nodes 12 and 14 is between 13t and 14t. Node 11 is scheduled to execute on processor 3 at 8t.

Execution durations for allocations of the graph in figure 20, for token communication times of 0, t/2 and t, are found to be 15t, 16t and 18t respectively. The minimum time of execution of the graph, as found by the earliest time analysis, is 15t.

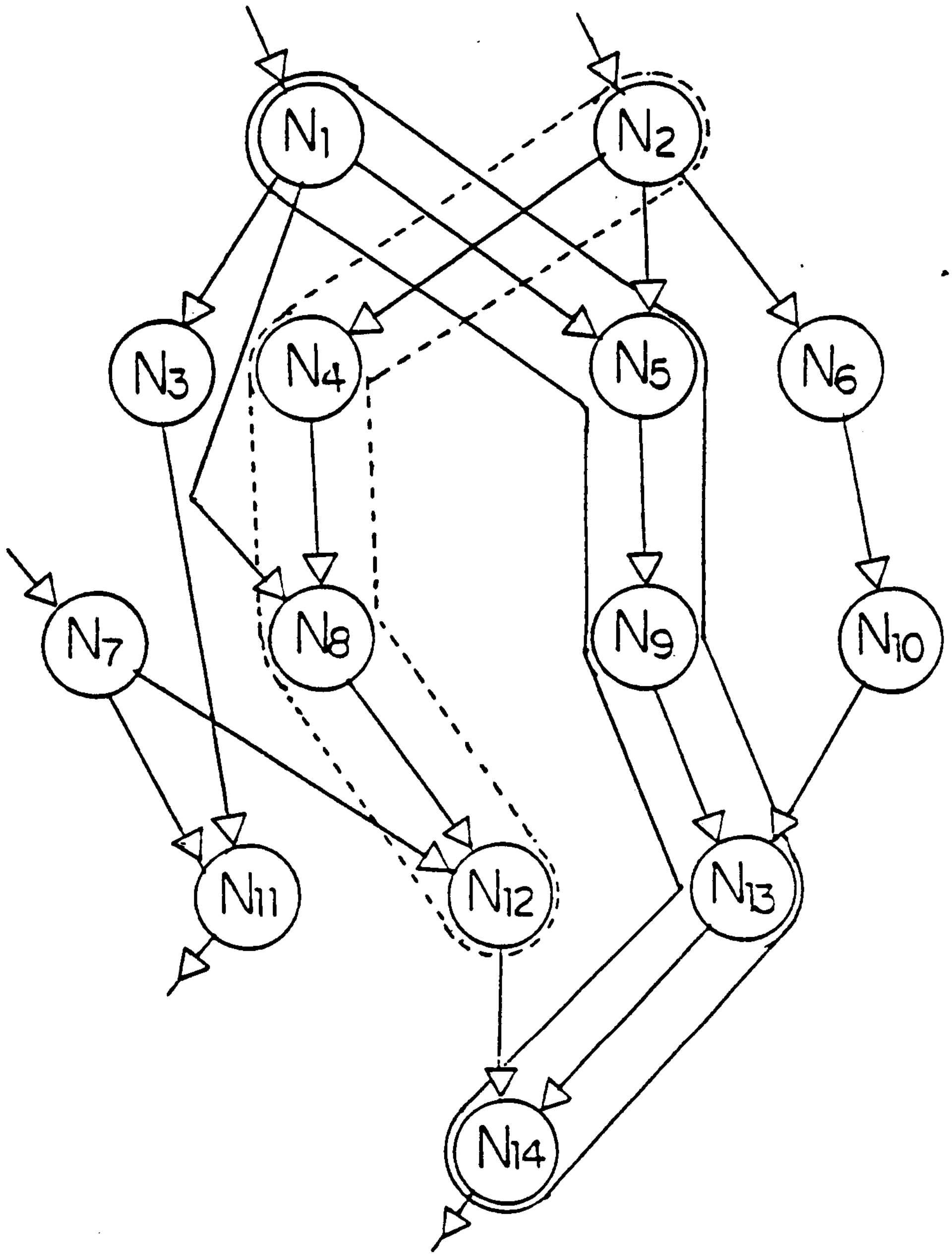

Figure 20. Program graph 2

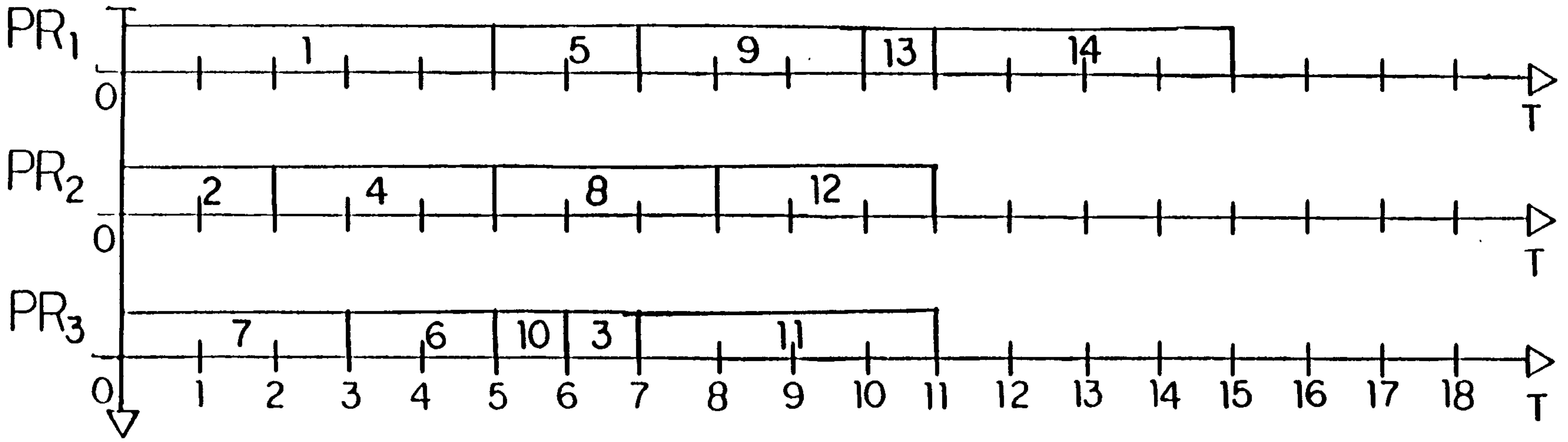

Figure 21. Timing diagram of graph represented by table 13 for CT = 0

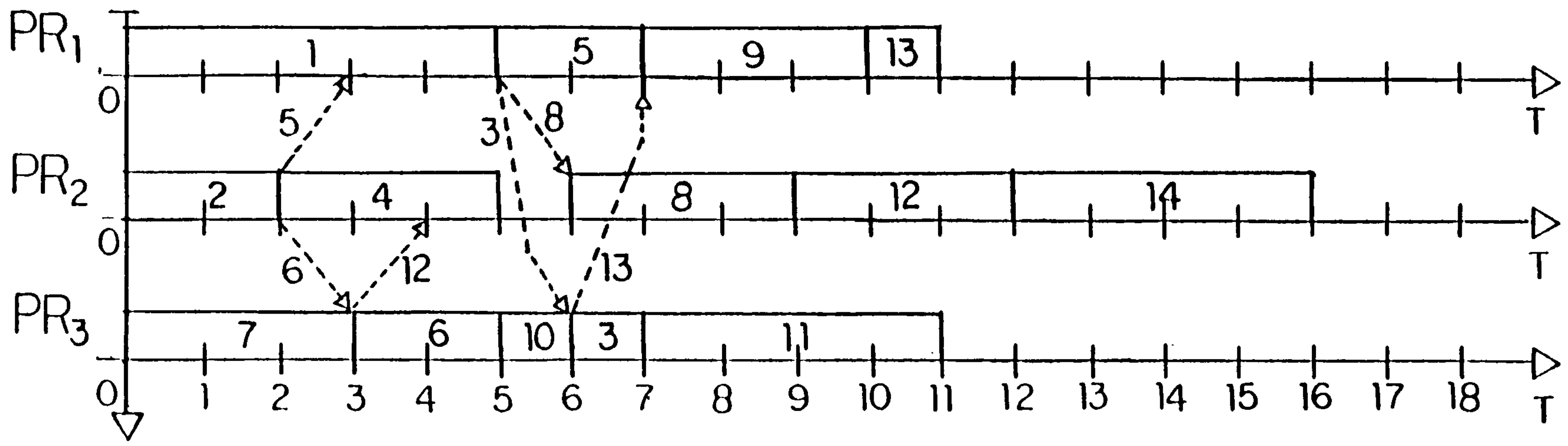

Figure 22. Timing diagram of graph represented by table 13 for CT = t/2

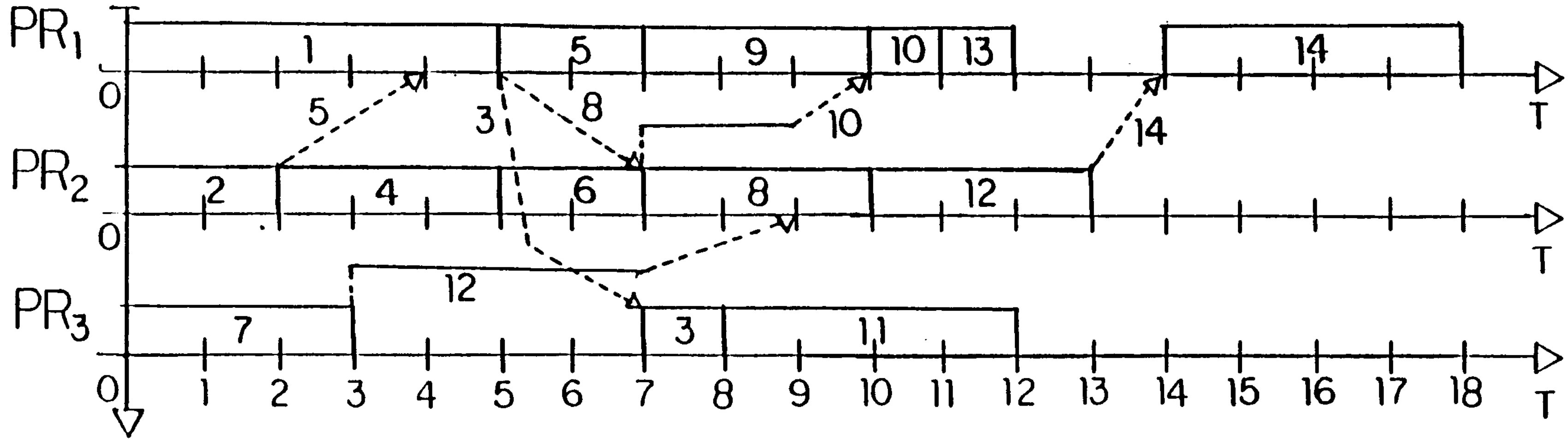

Figure 23. Timing diagram of graph represented by table 13 for CT = t

Table 13
Pre-allocation analysis for figure 20

Node Number	Execution Duration	Number of Tokens	Earliest Time	Latest Time	Order
1	5t	2	0	0	1
2	2t	2	0	0	2
3	t	1	5t	10t	12
4	3t	1	2t	2t	3
5	2t	1	5t	5t	4
6	2t	1	2t	7t	8
7	3t	2	0	5t	6
8	3t	1	5t	5t	5
9	3t	1	7t	7t	7
10	t	1	4t	9t	10
11	4t	1	6t	11t	14
12	3t	1	8t	8t	9
13	t	2	10t	10t	11
14	4t	1	11t	11t	13

Table 14
Resource allocation for table 13 when CT = 0

Processor	Nodes (x,y,z)
PR1	(0,1,5t) (5t,5,7t) (7t, 9,10t) (10t,13,11t) (11t,14,15t)
PR2	(0,2,2t) (2t,4,5t) (5t, 8, 8t) (8t,12,11t)
PR3	(0,7,3t) (3t,6,5t) (5t,10, 6t) (6t, 3, 7t) (7t,11,11t)
Total execution duration = 15t; Minimum time of execution = 15t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 15
Resource allocation for table 13 when CT = t/2

Processor	Nodes (x,y,z)
PR1	(0,1,5t) (5t,5,7t) (7t, 9,10t) (10t,13,11t) (12.5t,14,16.5t)
PR2	(0,2,2t) (2t,4,5t) (6t, 8, 9t) (9t,12,12t)
PR3	(0,7,3t) (3t,6,5t) (5t,10, 6t) (6t, 3, 7t) (7.0t,11,11.0t)
Total execution duration = 16.5t; Minimum time of execution = 15t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 16
Schedule of interprocessor communication for table 13 when CT = t/2

Source node	Processor & exec. time	# of tokens	Schedule of communication (d.p,x,y)
N1	PR1,0,5t	2	(N3,PR3,5t,6t) (N8.PR2,5t.6t)
N2	PR2,0,2t	2	(N5,PR1,2t,3t) (N6,PR3,2t,3t)
N3	PR3,6t,7t	1	
N4	PR2,2t,5t	1	
N5	PR1,5t,7t	1	
N6	PR3,3t,5t	1	
N7	PR3,0,3t	2	(N12,PR2,3t,4t)
N8	PR2,6t,9t	1	
N9	PR1,7t,10t	1	
N10	PR3,5t,6t	1	(N13,PR1,6t,6.5t)
N11	PR3,7t,11t	1	
N12	PR2,9t,12t	1	
N13	PR1,10t,11t	2	(N14,PR2,11t,12t)
N14	PR2,12t,16t	1	

(d,p,x,y) : Communication to dependent 'd' allocated to processor 'p' is scheduled between 'x' and 'y'.

Table 17
Resource allocation for table 13 when CT = t

Processor	Nodes (x,y,z)
PR1	(0,1,5t) (5t,5,7t) (7t, 9,10t) (10t,10,11t) (11t,13,12t) (14t,14,18t)
PR2	(0,2,2t) (2t,4,5t) (5t, 6, 7t) (7t, 8,10t) (10t,12,13t)
PR3	(0,7,3t) (7t,3,8t) (8t,11,12t)
Total execution duration = 18t; Minimum time of execution = 15t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

Table 18
Schedule of interprocessor communication for table 13 when CT = t

Source node	Processor & exec. time	# of tokens	Schedule of communication (d,p,x,y)
N1	PR1,0,5t	2	(N3,PR3,5t,7t) (N8,PR2,5t,7t)
N2	PR2,0,2t	2	(N5,PR1,2t,4t)
N3	PR3,7t,8t	1	
N4	PR2,2t,5t	1	
N5	PR1,5t,7t	1	
N6	PR2,5t,7t	1	(N10,PR1,9t,10t)
N7	PR3,0,3t	2	(N12,PR2,7t,9t)
N8	PR2,7t,10t	1	
N9	PR1,7t,10t	1	
N10	PR1,10t,11t	1	
N11	PR3,8t,12t	1	
N12	PR2,10t,13t	1	(N14,PR1,13t,14t)
N13	PR1,11t,12t	2	
N14	PR1,14t,18t	1	

(d,p,x,y) : Communication to dependent 'd' allocated to processor 'p' is scheduled between 'x' and 'y'.

CHAPTER FOUR

RESULTS AND CONCLUSIONS

This research has produced expected results. Given a sufficient number of processing elements, on the condition that interprocessor token communication time is zero, the algorithm has been successful in spreading any program graph over multiple processors to obtain the 'minimum time of execution' as determined by the earliest time analysis.

For non-zero values of interprocessor communication, the increase in total execution duration has been found to be small compared to the increase in interprocessor token communication times. This is due to the effort made in scheduling data-dependent nodes to the same processor, and, considering both interprocessor communication time and execution duration in assigning a node to the same, or to a different processor, as its parent. The number of interprocessor communications scheduled by the algorithm decreased with increased interprocessor token-communication durations; i.e., more nodes were scheduled to a single processor so as to draw a balance between communication and queueing delays. The number of processors over which a program graph was spread, decreased slightly with increased communication durations. Furthermore, even amongst those used, increased communication durations also resulted in extra utilization of some processors as compared to others.

The allocation made and the resulting execution duration , varied with the architecture of the communication network. Although, allocation of single sub-functions which are mostly sequential did not result in much difference in the execution durations, allocation of multiple sub-functions with little data-dependency, resulted in higher execution durations for a blocking communication network than it did for a non-blocking crossbar. This suggests, sub-functions should be allocated to different sub-sections of the arhchitecture, having independent communication structures. When allocated to a common sub-section, the communication structure has to be non-blocking to allow simultaneous execution of sub-functions.

Tables 14, 15 and 17 represent the allocations of the graph represented by table 13 for token communication durations of 0, t/2 and t respectively. Tables 16 and 18 represent the corresponding communication schedules for token communication durations of t/2 and t. The minimum time of execution of this graph is 15t. Allocations made for communication times of 0, t/2 and t would require 15t, 16.5t and 18t time units respectively. The serial execution duration of this graph is 37t. Hence, the resulting speedup factor, which is the ratio of sequential and parallel execution durations, was found to be 2.47, 2.24 and 2.06 respectively, for interprocessor communication durations of 0, t/2 and t.

When both program graphs of figures 13 and 20 were allocated to a single sub-section, having a non-blocking communication network, they still simultaneously completed execution within the same time frame, as independent allocations did in different sub-sections. However, when the crossbar was replaced with a blocking network, total execution duration of the two graphs was found to be 32t, almost equal to the sum of their individual execution durations.

This indicates that parallelism between sub-functions could not be exploited in the latter case, due to limitations of the communication structure.

The program graph for a missile guidance problem being analyzed at the University of Alabama is represented in table A of appendix A. It has 109 nodes, with varying node execution durations (upto 110t). The minimum time of execution of this graph has been determined to be 300t time units. Considering token communication duration to be zero, the allocation made by the algorithm is represented in table B of appendix A. This graph was spread on 30 processors, and executed in 300t time units. As the total sequential execution time of this graph was 6500t, the speedup factor was found to be 21.66, and the reduction in execution duration was 6200t. With communication durations considered equal to execution durations of parent nodes, the graph was scheduled over 26 processors to execute within a time frame of 480t. This resulted in a reduction of 6020t in the execution duration, and a speedup factor of 13.54. The speedup factor was therefore found to reduce in finely granular graphs, where the interprocessor communication time plays a significant role. In other words, practical systems with significant finite values of interprocessor communication times, are expected to return a lower value of the speedup factor as compared to the ideal situation.

BIBLIOGRAPHY

1. Yang-Chang Hong, Thomas H. Payne and Le Baron O. Ferguson, "Graph Allocation in Static Dataflow Systems," Proceedings of IEEE, 1986, pp. 55-64.

2. Michael L. Campbell, "Static Allocation for a Dataflow Multiprocessor," Proceedings of IEEE on Parallel Processing, 1985, pp. 511-517.

3. L.Y. Ho and K.B. Irani, "An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment," Proceedings of theInternational Conference on Parallel Processing, 1983, pp. 338-340.

4. Pauline Markenscoff and Weikuo Liaw, "Task Allocation Problems in Distributed Computer Systems," IEEE Proceedings, 1986, pp. 953-960.

5. K.J. Mundell, M.W. Linder and S.E. Conry, "Processor Allocation in Data Driven Systems - Two Approaches," Proceedings of the International Conference on Parallel Processing, 1981, pp. 156-157.

6. Y.C. Hong, T.H. Payne and L.O. Ferguson, "Partitioning Program Graphs to Enhance Concurrency in Static Dataflow Systems," Dept. of Math and Computer Science, University of California, Riverside, November 1985.

7. D. Johnson, "Automatic Partitioning of Programs in Multiprocessor Systems," IEEE Compson, 1985.

8. C.C. Price, "The Assignment of Computational TAsks Among Processors in a Distributed System," AFIPS Conference Proceedings, Vol.50, 1981.

9. C.C Shen and W.H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," IEEE Transactions on Computers, Vol. C-34, No. 3, March 1985.

10. John Gurd and Ian, "Data Driven System for High Speed Parallel Computing," Computer Design, Jul. 1980, pp. 97-106.

11. L.M. Patnaik and Julie Basu, "Two Tolls for Interprocess Communication in Distributed Dataflow systems," Computer Journal, Vol. 29, No. 6, 1986, pp. 506-521.

12. Allan Gottlieb and J.T Schwartz, "Networks and Algorithms for Very-Large-Scale Parallel Computation," Computer, Jan. 1982, pp. 27-36.

13. Daniel Gajski, David Kuck, Duncan Lawrie and Ahmed Sameh, "Cedar- A Large Scale Multiprocessor," Proceedings of International Conference on Parallel Processing, 1983, pp. 524-529.

14. R. Vedder, M. Campbell, and G. Tucker, "The Hughes Dataflow Multiprocessor," Microelectronics Engineering Laboratory, Technical Report, 1985, pp. 2-9.

15. M. Amamiya and R. Hasegawa, "Parallel Execution of Logic Programs Based on Dataflow Concept," Fifth Generation Computer Systems 1984, Proceesings of the International Conference on Fifth Generation Computer Systems, 1984, pp. 507-516.

16. J.D Ullman, "Polynomial NP-Complete Scheduling Problems," Operating Systems Review, Vol. 7, No. 4, 1973, pp. 96-101.

17. T.L. Adam, K.P. Chandy and J.R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," Communication ACM, Vol. 17, Dec. 1974, pp. 685-690.

18. H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," IEEE Transactions on Computers, Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

APPENDIX A

Table A
Pre-allocation analysis of the missile guidance problem

Node #	Exec. Time	# of Tokens	Earliest Time	Latest Time	Order	Parents	Dependents
1	0		0	0	0		2,3,4,5
2	30t		0	40t	28	1	6,45,46,57,58,69,70,77 78,95,96,105,106
3	50t		0	80t	30	1	49,50,63,64,75,76,81 82,101,102
4	20t		0	0	1	1	7,8,...,108,109
5	20t		0	130t	42	1	87,88,91,92,97,98,103
6	60t		30t	70t	29	2	35,36,37,38,39,40,83,84
7	110t		20t	90t	30	4	43
8	110t		20t	90t	31	4	44
9	110t		20t	40t	16	4	45,55,87
10	110t		20t	40t	17	4	46,56,88
11	110t		20t	40t	18	4	49,61
12	110t		20t	40t	19	4	50,62
13	110t		20t	40t	20	4	35,41,45,49,67
14	110t		20t	40t	21	4	36,42,46,50,68
15	110t		20t	40t	22	4	61,91
16	110t		20t	40t	23	4	62,92
17	110t		20t	40t	24	4	67,69,97
18	110t		20t	40t	25	4	68,70,98
19	110t		20t	20t	2	4	37,43,53,57,63,77,103
20	110t		20t	20t	3	4	38,44,54,58,64,78,104
21	110t		20t	20t	4	4	75
22	110t		20t	20t	5	4	76
23	110t		20t	20t	6	4	39,47,59,69,71.75,79,81
24	110t		20t	20t	7	4	40,48,60,70,72,76,80,82
25	110t		20t	20t	8	4	51,65,77,81,83,105
26	110t		20t	20t	9	4	52,66,78,82,84,106
27	110t		20t	40t	26	4	91
28	110t		20t	40t	27	4	92
29	110t		20t	20t	10	4	95
30	110t		20t	20t	11	4	96
31	110t		20t	20t	12	4	101
32	110t		20t	20t	13	4	102
33	110t		20t	20t	14	4	89,93,95,99,101,107
34	110t		20t	20t	15	4	90,94,96,100.102,108
35	30t		130t	170t	69	6,13	41,43,47,51,89,93
36	30t		130t	170t	70	6,14	42,44,48,52,90,94
37	30t		130t	130t	32	6,19	53,59,65

Table A contd.

Node #	Exec. Time	# of Tokens	Earliest Time	Latest Time	Order	Parents	Dependents
38	30t		130t	130t	33	6,20	54,60,66
39	30t		130t	150t	43	6,23	71,79,99
40	30t		130t	150t	44	6,24	72,80,100
41	30t		160t	220t	105	13,35	109
42	30t		160t	220t	106	14,36	109
43	50t		160t	200t	95	7,19,35	109
44	50t		160t	200t	96	80,20,36	109
45	50t		130t	150t	45	2,9,13	47
46	50t		130t	150t	46	2,10,14	48
47	50t		180t	200t	93	23,35,45	109
48	50t		180t	200t	94	24,36,46	109
49	50t		130t	150t	47	3,11,13	51
50	50t		130t	150t	48	3,12,14	52
51	50t		180t	200t	89	25,35,49	109
52	50t		180t	200t	90	26,36,50	109
53	30t		160t	180t	79	19,37	55
54	30t		160t	180t	80	20,38	56
55	40t		190t	210t	99	9,53	109
56	40t		190t	210t	100	10,54	109
57	30t		130t	160t	60	2,19	61
58	30t		130t	160t	61	2,20	62
59	30t		130t	160t	62	23,37	61
60	30t		160t	160t	57	24,38	62
61	60t		160t	190t	88	11,15,57,59	109
62	60t		190t	190t	85	12,16,58,60	109
63	30t		130t	160t	63	3,19	67
64	30t		130t	160t	64	3,20	68
65	30t		160t	160t	58	25,37	67
66	30t		160t	160t	59	26,38	68
67	60t		190t	190t	86	13,17,63,65	109
68	60t		190t	190t	87	14,18,64,66	109
69	50t		130t	160t	65	2,17,23	73
70	50t		130t	160t	66	2,18,24	74
71	30t		160t	180t	81	23,39	73
72	30t		160t	180t	82	24,40	74
73	40t		190t	210t	101	69,71	109
74	40t		190t	210t	102	70,72	109
75	50t		130t	130t	34	3,21,23	79
76	50t		130t	130t	35	3,22,24	80
77	50t		130t	130t	36	2,19,25	79
78	50t		130t	130t	37	2,20,26	80
79	70t		180t	180t	73	23,39,75,77	109
80	70t		180t	180t	74	24,40,76,78	109
81	50t		130t	160t	67	3,23,25	85
82	50t		130t	160t	68	3,24,26	86
83	30t		130t	180t	83	6,25	85
84	30t		130t	180t	84	6,26	86
85	40t		180t	210t	103	81,83	109

Table A contd.

Node #	Exec. Time	# of Tokens	Earliest Time	Latest Time	Order	Parents	Dependents
86	40t		180t	210t	104	82,84	109
87	30t		130t	170t	71	5,9	89
88	30t		130t	170t	72	5,10	90
89	50t		160t	200t	97	33,35,87	109
90	50t		160t	200t	98	34,36,88	109
91	50t		130t	150t	49	5,15,27	93
92	50t		130t	150t	50	5,16,28	94
93	50t		180t	200t	91	33,35,91	109
94	50t		180t	200t	92	34,36,92	109
95	50t		130t	130t	38	2,29,33	99
96	50t		130t	130t	39	2,30,34	100
97	30t		130t	150t	51	5,17	99
98	30t		130t	150t	52	5,18	100
99	70t		180t	180t	75	33,39,97,95	109
100	70t		180t	180t	76	34,40,98,96	109
101	50t		130t	130t	40	3,31,33	107
102	50t		130t	130t	41	3,32,34	108
103	30t		130t	150t	53	5,19	107
104	30t		130t	150t	54	5,20	108
105	30t		130t	150t	55	2,25	107
106	30t		130t	150t	56	2,26	108
107	70t		180t	180t	77	33,101,103,105	109
108	70t		180t	180t	78	34,102,104,106	109
109	50t		250t	250t	107	41...44,47,48,51 52,55,56,61,62,67 68,73,74,79,80,85 86,89,90,93,94,99 100,107,108	110
110	0		300t	300t	110	109	

Table B
Resource allocation for Table A when CT = 0

Processor	Nodes
PR1	(0,4,20t) (20t,19,130t) (130t,37,160t) (160t,65,190t) (190t,67,250t) (250t,109,300t)
PR2	(0,5,20t) (20t,20,130t) (130t,38,160t) (160t,60,190t) (190t,62,250t)
PR3	(20t,21,130t) (130t,75,180t) (180t,79,250t)
PR4	(20t,22,130t) (130t,76,180t) (180t,80,250t)
PR5	(20t,23,130t) (130t,39,160t) (160t,59,190t) (190t,61,250t)
PR6	(20t,24,130t) (130t,40,160t) (160t,63,190t) (190t,47,240t)
PR7	(20t,25,130t) (130t,77,160t) (160t,87,190t) (190t,51,240t)
PR8	(20t,26,130t) (130t,78,160t) (160t,66,190t) (190t,68,250t)
PR9	(20t,29,130t) (130t,103,160t) (160t,88,190t) (190t,48,240t)
PR10	(20t,30,130t) (130t,96,180t) (180t,100,250t)
PR11	(20t,31,130t) (130t,101,180t) (180t,107,250t)
PR12	(20t,32,130t) (130t,102,180t) (180t,108,250t)
PR13	(20t,33,130t) (130t,95,180t) (180t,99,250t)
PR14	(20t,34,130t) (130t,104,160t) (180t,94,230t)
PR15	(20t,9,130t) (130t,45,180t) (180t,71,210t) (210t,55,250t)
PR16	(20t,10,130t) (130t,46.180t) (180t,72,210t) (210t,56,250t)
PR17	(20t,11,130t) (130t,49.180t) (180t,83,210t) (210t.41,240t)
PR18	(20t,12,130t) (130t,50,180t) (180t,84,210t) (210t,86,250t)
PR19	(20t,13,130t) (130t,105.160t) (160t,35,190t) (190t,93,240t)
PR20	(20t,14,130t) (130t,106,160t) (160t,36,190t) (190t,52.240t)
PR21	(20t,15,130t) (130t,91,180t) (180t,53,210t) (210t,42,240t)
PR22	(20t,16,130t) (130t,92,180t) (180t,54,210t)
PR23	(20t,17,130t) (130t,97,160t) (160t,64,190t) (190t,43,240t)
PR24	(20t,18,130t) (130t,98,160t) (160t,69,210t) (210t,73,250t)
PR25	(20t,27,130t) (130t,57,160t) (160t,70,210t) (210t,74,250t)
PR26	(20t,28,130t) (130t,58,160t) (160t,81,210t) (210t,85,250t)
PR27	(0,2,30t) (30t,6,90t) (90t,7,200t) (200t,44,250t)
PR28	(0,3,50t) (50t,8,160t) (160t,82,210t)
PR29	(160t,89,210t)
PR30	(160t,90,210t)
Total execution duration = 300t; Minimum time of execution = 300t	

(x,y,z) : Node N_y is scheduled to begin execution at x and end at z

APPENDIX B

ALGORITHM FOR THE DETERMINATION OF EARLIEST TIMES OF ALL NODES.

- Starting with the root node (Node # 1 which has no parents), and continuing until the final node.

- Earliest time of any node is the latest of 'the earliest times of its parents plus their respective execution durations'.

Assuming N_a, N_b and N_c to be the parents of N_x. Let X_a, X_b and X_c be the execution durations and E_a, E_b and E_c be the earliest times of N_a, N_b and N_c respectively. Then, the earliest time of N_x is the largest of

$$((E_a + X_a), (E_b + X_b), (E_c + X_c))$$

- Finally the 'minimum time of execution' of the graph is determined.

APPENDIX C

ALGORITHM FOR THE DETERMINATION OF THE LATEST TIMES OF ALL NODES.

- Starting with the final node.

- Latest time of nodes with no parents is equal to the difference between the minimum time of execution and the execution duration of the node.

- Latest time of other nodes is the minimum of 'the latest times of the dependents minus the execution duration of the node'.

Assuming N_a, N_b and N_c to be the dependents of N_X and L_a, L_b and L_c their respective latest times, and X the execution duration of N_X, the latest time of N_X is given by the minimum of

$$((L_a - X), (L_b - X), (L_c - X))$$

- Finally critical nodes are identified.

APPENDIX D

```
Program Allocate;

type

 (* Each node has associated with it a record called node
   information, which has all the necessary information
   concerning that node *)

 node_information = record
        instruction_number : integer;
        execution_duration : real;
        number_of_sources : integer;
        number_of_destinations : integer;
        source_nodes : array (.1..5.) of integer;
        destination_nodes : array (.1..5.) of integer;
        earliest_time : real;
        latest_time : real;
        critical : boolean;
        output_priority : array (.1..5.) of integer;
        processor_allocated : integer;
        load_time : real;
        allocated : boolean;
      end;

 (* The dataflow graph has many nodes in it. *)

 dataflow_graph = array (.1..100.) of node_information;

 (* Within the record of each processor is a field which is
   a array of pointer records. These pointer records
   'exec_time_ranges' contain the beginning and ending
   times of the execution of a particular instruction, and the
   node number of that instruction. ')

 exec_time_pointer = @ exec_time_ranges;
 exec_time_ranges = record
                 exec_time_begin : real;
                 exec_time_end : real;
                 exec_node : integer;
                 exec_link : exec_time_pointer;
                end;

(* Also, within the record of each processor, is a structure similar
  to 'exec_time_ranges'. This structure called the 'comm_time_ranges
```

```
    keeps the beginning and the ending times of communication to and
    from the processor, direction of communication, and the number of
    the other processor that is involved in the communication. *)

comm_time_pointer = @ comm_time_ranges;
comm_time_ranges = record
                    comm_time_begin : real;
                    comm_time_end : real;
                    origin_node : integer;
                    target_node : integer;
                    target_processor : integer;
                    comm_direction : boolean;
                    comm_time_link : comm_time_pointer;
                  end;

(* Associated with each processor is a record which houses the
   pertinent information. The 'final_ready_time' is the time
   beyond which no nodes are allocated to the processor for
   execution. 'Busy_times' and 'communication_times' give
   information about the time ranges when the processor is
   executing and communication respectively. The nodes allocated
   to the processor are stored in an array. *)

processor_information  = record
                          number_of_nodes_allocated : integer;
                          final_ready_time : real;
                          busy_times : exec_time_ranges;
                          communication_times : comm_time_ranges;
                          head_busy : exec_time_pointer;
                          head_communication : comm_time_pointer;
                          nodes_allocated : array (.1..100.) of
                                                integer;
                        end;

(* All the processors form a processor list. The execution times
   of the various instructions are stored in an array. *)

processor_list = array (.1..20.) of processor_information;
table_of_execution_times = array (.1..20.) of real;

(* Each critical path is a variable length array of records. Each of
   these records contains the serial number and the actual number of
   of a node that belongs to this critical path. There is also a
   pointer to the next of such records if any. All the critical path
   in the dataflow graph are grouped together in 'critical_path_
   collection' which is a structure similar to a 'critical_path'.
   This structure contains one record for each critical_path in the
   dataflow graph. Each of these records contains a 'critical_path'
   structure, its serial number, execution_duration, and a pointer
   to the next of such records housing information about the next
   critical_path. *)

critical_pointer = @ critical_path;
critical_path = record
                 node_serial_number : integer;
```

```
                  node_number : integer;
                  next_node : critical_pointer;
                end;
  critical_paths_pointer = @ critical_path_collection;
  critical_path_collection = record
                              path_serial_number : integer;
                              current_path : critical_path;
                              path_exec_duration : real;
                              head_current_path : critical_pointer;
                              next_path : critical_paths_pointer;
                            end;


var

 node : dataflow_graph;
 processor : processor_list;
 instruction_duration : table_of_execution_times;
 network_communication_times : comm_time_ranges;
 head_network_communication : comm_time_pointer;
 critical_collection : critical_path_collection;
 head_critical_collection : critical_paths_pointer;
 number_of_nodes : integer;
 earliest_graph_execution : real;
 allocation_file,filename : string(.15.);
 infile,outfile : text;


 (* Procedure execution_times checks the instruction number associated
    with each node, and after refering to the table of instruction_
    duration updates the execution_duration nodal information. *)


 Procedure Execution_times ( var node : dataflow_graph;
                var instruction_duration : table_of_execution_times;
                number_of_nodes : integer );

 var

  instruction, node_number : integer;


 begin

  for node_number := 1 to number_of_nodes do
    begin
      instruction := node(.node_number.).instruction_number;
      node(.node_number.).execution_duration :=
                          instruction_duration (.instruction.);
    end; (* for *)

 end; (* procedure execution_times *)
```

```
(* Procedure earliest_times calculates the earliest time at which
   each node can be executed after satisfying the dependency
   constraints, and updates the corresponding earliest time nodal
   information.  The earliest a node can execute, is the earliest
   time when all the information has arrived to it from its parents
   when they have also executed at their respective earliest times.
   Thus the earliest time of a node is the latest of 'the earliest
   times of its parents plus their respective execution durations'.
   'Earliest_graph_execution' is the earliest time that a graph may
   execute when all the nodes in it have executed at their earliest
   times. The earliest time of a node with no parents is zero. *)

Procedure Earliest_times ( var node : dataflow_graph ;
                           var earliest_graph_execution : real;
                           number_of_nodes : integer );

var

  earliest, current_earliest : real;
  sources, node_number, current_source : integer;

begin

  for node_number := 1 to number_of_nodes do
    begin
      earliest := 0;
      for sources := 1 to node (.node_number.).number_of_sources do
        begin
          current_source := node(.node_number.).source_nodes(.sources.);
          current_earliest := (node(.current_source.).earliest_time +
                          node(.current_source.).execution_duration);
          if (current_earliest > earliest) then
             earliest := current_earliest;
        end; (* inner for *)
      node(.node_number.).earliest_time := earliest;
      if (earliest > earliest_graph_execution) then
         earliest_graph_execution := earliest;
    end; (* outer for *)

end; (* procedure earliest_times *)


(* Procedure Latest_times determines the latest time that each node
   may execute without delaying the total graph execution beyond its
   earliest time 'earliest_graph_execution'.  The latest time of a
   node with no dependents is the earliest_graph_execution minus its
   execution duration.  The latest time of a node with dependents is
   the earliest of 'the latest times of its dependents minus their
   respective execution durations'. A node is said to be critical if
   its earliest time is equal to its latest time. *)

Procedure Latest_times ( var node : dataflow_graph;
                         number_of_nodes : integer;
                         earliest_graph_execution : real);
```

```
var

  node_number, destination, current_destination : integer;
  latest, current_latest : real;


begin

  for node_number := number_of_nodes downto 1 do
    begin
      latest := (earliest_graph_execution -
                   node (.node_number.).execution_duration);
      for destination := 1 to node(.node_number.).number_of_destinations
        do begin
          current_destination :=
             node(.node_number.).destination_nodes(.destination.);

          current_latest := node(.current_destination.).latest_time -
                         node(.node_number.).execution_duration;
          if (current_latest < latest) then
             latest := current_latest;
        end; (* inner for *)
      node(.node_number.).latest_time := latest;
      if (latest = node(.node_number.).earliest_time) then
         node(.node_number.).critical := true
      else node(.node_number.).critical := false;
    end; (* outer for *)

end; (* procedure latest_times *)


(* Procedure Check_dependency is invoked by procedure Update_path only.
   It checks whether 'node_num' is a dependent of 'current_node_num',
   and returns a true value in the boolean variable 'member_of_current_
   path' if this is true. This is used in determining whether 'node_num'
   is a member of the critical_path that is being scanned. *)

Procedure Check_dependency ( var node : dataflow_graph;
                             current_node_number, node_num : integer;
                             var member_of_current_path : boolean);


var

  dependent : boolean;
  dep_num : integer;

begin

  dependent := false;
  dep_num := 1;
  while (dep_num < = node(.current_node_number.).number_of_destinations)
      and (dependent = false) do
    begin
      if (node(.current_node_number.).destination_nodes(.dep_num.) =
        node_num) then
      begin
        dependent := true;
        member_of_current_path := true;
      end; (* if *)
    dep_num := dep_num + 1;
  end; (* while *)

end; (* Procedure Check_dependency *)

(* Procedure Update_path is invoked by procedure Critical_paths only.
   It checks whether 'node_num' should belong to the critical path
   'path', and if so adds it on to the list of nodes in this critical
   path.  It invokes procedure Check_dependency to check whether
   'node_num' is a dependent of any of the nodes that were earlier
   entered in path's list.  If this is found to be true, 'node_num'
   is considered to be a member of 'path', and the boolean variable
   'found' is returned as true.  *)

Procedure Update_path ( var path : critical_path;
                        var path_exec_duration : real;
                        head : critical_pointer;
                        var node : dataflow_graph;
                        node_num : integer;
                        var found : boolean );

var

  current_node, create_node : critical_pointer;
  member_of_current_path : boolean;
  current_node_number : integer;

begin

  member_of_current_path := false;
  current_node := head;

  (* The while loop given below goes through the pointer list of
     nodes that are already identified as being members of the
     critical path passed to this procedure.  Procedure Check_
     dependency is called in an iteration if 'node_num' has not
     been identified as a member of the critical path.  Finally
     if 'node_num' has been identified as a member of the critical
     path, it is joined to the list of the path's members at the
     end, and 'found' is returned as true. *)

  while (current_node < > nil) do
    begin
      if (member_of_current_path = false ) then
        begin
          current_node_number := current_node @.node_number;
          Check_dependency (node,current_node_number,node_num,
                    member_of_current_path);
        end; (* if *)
      if (current_node @.next_node = nil) then
```

```
        begin
          if (member_of_current_path = true) then
            begin
              found := true;
              New (create_node);
              current_node @.next_node := create_node;
              create_node @.next_node := nil;
              create_node @.node_number := node_num;
              create_node @.node_serial_number := 1 +
                 current_node @.node_serial_number;
              path_exec_duration := path_exec_duration +
                 node(.node_num.).execution_duration;
            end; (* inner if *)
          current_node := nil;
        end (* outer if *)
      else
        current_node := current_node @.next_node;
    end; (* while *)

end; (* Procedure Update_path *)


(* Procedure Critical_paths determines the critical paths that are
   present in the program graph.  It scans through the nodes of the
   graph, and for those that are critical, it determines whether they
   are a part of the earlier determined paths by repeatedly invoking
   procedure Update_path.  If the critical node is not found to be a
   part of earlier determined paths, then a new path is created with
   the current critical node as its first ,member.  *)

Procedure Critical_paths ( var node : dataflow_graph;
               var critical_collection : critical_path_collection;
               var head_critical_collection : critical_paths_pointer;
               var number_of_critical_paths : integer );

var

  examined_path, previous_path, created_path : critical_paths_pointer;
  node_num : integer;
  found : boolean;

begin

  number_of_critical_paths := 0;
  head_critical_collection := nil;
  for node_num := 1 to number_of_nodes do
    if (node(.node_num.).critical = true) then
      begin
        examined_path := head_critical_collection;
        found := false;
        (* The while loop below checks wheter 'node_num' is a member
           of the various critical paths already identified, and if so
           updates those that contain this node.  *)
        while (examined_path < > nil) do
          begin
          Update_path (examined_path @.current_path,
                examined_path@.path_exec_duration,
                examined_path @.head_current_path,
                nodes,node_num,found);
          examined_path := examined_path @.next_path;
        end; (* while *)
      (* If the critical node 'node_num' was not found in the critical
         paths in the while loop above, then a new critical path is
         created in the if structure below, and added to the end in
         the list of critical paths.  *)
      if (found = false) then
        begin
          number_of_critical_paths := 1 + number_of_critical_paths;
          previous_path := head_critical_collection;
          New(created_path);
          while (previous_path < > nil) do
            if (previous_path @.next_path = nil) then
              begin
                created_path@.path_exec_duration := 0;
                previous_path @.next_path := created_path;
                created_path @.path_serial_number := 1 +
                   previous_path @.path_serial_number;
                previous_path := nil;
              end (* if *)
            else
              previous_path := previous_path @.next_path;
          if (head_critical_collection = nil) then
            begin
              head_critical_collection := created_path;
              created_path @.path_serial_number := 1;
            end;
          created_path @.next_path := nil;
          New(created_path @.head_current_path);
          created_path @.head_current_path @.node_serial_number := 1;
          created_path @.head_current_path @.next_node := nil;
          created_path @.head_current_path @.node_number := node_num;
        end; (* if found = false *)
    end; (* if node(.node_num.).critical = true *)

end;  (* procedure Critical_paths *)



Procedure Determine_processor ( var node : dataflow_graph;
          var processor : processor_list;
          var instruction_duration : table_of_execution_times;
          var network_communication_times : comm_time_ranges;
          var head_network_communication : comm_time_pointer;
          node_number : integer;
          var processor_selected : integer;
          critical : boolean;
          processor_to_be_allocated : integer;
          number_of_critical_paths : integer );
```

```
begin

  head_network_communication := nil;
  earliest := node(.node_number.).earliest_time;
  latest := node(.node_number.).latest_time;
  command_number := node(.node_number.).instruction_number;
  exec_time_duration := instruction_duration(.command_number.);
  node(.node_number.).execution_duration := exec_time_duration;
  if (critical = true) then
    begin
      if (processor_to_be_allocated > = 0) then
        begin
          if (processor(.processor_to_be_allocated.).final_ready_time
               > = earliest) then


Procedure Allocate_node_to_processor (var node : dataflow_graph;
   var processor : processor_list;
   node_num, processor_num : integer;
   time : real;
   var network_communication_times : comm_time_ranges;
   var head_network_communication : comm_time_pointer);

begin

  node(.node_num.).allocated := true;
  node(.node_num.).processor_allocated := processor_num;
  index := processor(.processor_num.).number_of_nodes_allocated;
  processor(.processor_num.).number_of_nodes_allocated := index + 1;
  processor(.processor_num.).nodes_allocated(.index.) := node_num;
  time_begin := time;
  time_end := time + node(.node_num.).execution_duration;
  if (time_end > processor(.processor_num.).final_ready_time) then
     processor(.processor_num.).final_ready_time := time_end;
  busy := processor(.processor_num.).head_busy;
  prev_busy := busy;
  added := false;
  while (busy < > nil) and (added < > true) do
    begin
      if (busy@.exec_time_begin > = time_end) then
        begin
          New (create_busy);
          if (busy = prev) then
             head := create_busy
          else
             prev_busy@.exec_link := create_busy;
          create_busy@.exec_link := busy;
          create_busy@.exec_time_begin := time_begin;
          create_busy@.exec_time_end := time_end;
          create_busy@.exec_node := node_num;
          added := true;
        end; (* if *)
      if (busy < > prev) then
        prev_busy := prev_busy@.exec_link;
      busy := busy@.exec_link;
    end; (* while *)

 end; (* Procedure Allocate_node_to_processor *)




 Procedure Allocate_critical_path (
          var critical_collection : critical_path_collection;
          var pointer_to_path : critical_paths_pointer;
          var network_communication_times : comm_time_ranges;
          var head_network_communication : comm_time_pointer;
          var node : dataflow_graph;
          var processor : processor_list;
          processor_num : integer) ;


 var

   current_node, pointer_to_node : critical_pointer;
   current_node_number : integer;
   time : real;

 begin

   pointer_to_node := pointer_to_path@.head_current_path;
   current_node := pointer_to_node;
   repeat
     current_node_num := current_node@.node_number;
     time := node(.current_node_number.).earliest;
     Allocate_node_to_processor(node,processor,current_node_num,
       processor_num,time,network_communication_times,
       head_network_communication);
     current_node := current_node@.next_node;
   until (current_node = nil);

 end; (* Procedure Allocate_Critical_path *)



Procedure Do_the_allocation (var node : dataflow_graph;
         var processor : processor_list;
         var instruction_duration : table_of_execution_times;
         var network_communication_times : comm_time_ranges;
         var critical_collection : critical_path_collection;
         var head_network_communication : comm_time_pointer;
         head_critical_collection : critical_paths_pointer;
         number_of_nodes : integer;
         number_of_processors : integer;
         number_of_critical_paths : integer);

var

  next_critical_path : critical_paths_pointer;
```

```
   next_critical_node : critical_pointer;
   node_number,processor_num : integer;

 begin

    head_network_communication := nil;
    next_critical_path := head_critical_collection;
    processor_num := 1;
    while (next_critical_path < > nil) do
     begin
       Allocate_critical_path(critical_collection,next_critical_path,
        network_communication_times,head_network_communication,
        node,processor,processor_num);
       processor_num := processor_num + 1;
       next_critical_path := next_critical_path@.next_path;
     end;
       next_critical_node := next_critical_path @.head_current_path;
       while (next_critical_node < > nil ) do
         begin
           node_number := next_critical_node @.node_number;
           Determine_processor (node,processor,instruction_duration,
                 network_communication_times,head_network_communication,
                 node_number,processor_selected);
           Update (processor,node,node_number,processor_selected);
           next_critical_node := next_critical_node @.next_node;
         end; (* inner while loop *)
       next_critical_path := next_critical_path @.next_path;
     end;  (* outer while loop *)
    for node_number := 1 to number_of_nodes do
      if (node(.node_number.).allocated < > true)then
         begin
           Determine_processor (node,processor,instruction_duration,
                 network_communication_times,head_network_communication,
                 node_number,processor_selected);
           Update (processor,node,node_number,processor_selected);
         end; (* if *)

 end;  (* procedure do_the_allocation *)


 (* The main body of the program reads nodal information from a user
 specified file, and invokes the various procedures to perform the
 allocation. *)

 begin

   writeln ('ALLOCATOR AT WORK');
   write   ('Program graph file name?');
   readln (filename);
   assign (infile,filename);
   reset (infile);
   readln (infile, number_of_nodes);

   for I := 1 to number_of_nodes do
     begin
```

```
      with node (.I.)
        readln (instrucion_number,num_of_sources,Num_of_dependents);
      for J := 1 to node (.I.).num_of_sources do
        with node (.I.)
          read (source_nodes (.J.));
      for J := 1 to node (.I.).num_of_dependents do
        with node (.I.)
          readln (destination_nodes (.J.));
    end;  (* input of nodes *)

  Execution_times (node,instruction_duration,number_of_nodes);
  Earliest_times (node,earliest_graph_execution,number_of_nodes);
  Latest_times (node,number_of_nodes,earliest_graph_execution);
  Critical_paths (node,critical_collection,head_critical_collection,
                 number_of_critical_paths);
  Do_the_allocation (node,processor,instruction_duration,
                    network_communication_times,critical_collection,
                    head_network_communication,head_critical_collection
                    number_of_nodes,number_of_processors,
                    number_of_critical_paths);

  writeln ('Output file name?');
  readln (allocation_file);
  assign (outfile,allocation_file);
  rewrite (outfile);
  for I := 1 to number_of_nodes do
    begin
      write (outfile,I,'  ',node(.I.).processor_allocated,'  ');
      write (outfile,node(.I.).load_time,'  ');
      write (outfile,node(.I.).execution_duration,'  ');
    end;

  Close (infile);
  Close (outfile);

END.
```